TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Efficient secure computation of AES

Papachristodoulou, L.

Award date:
2010

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 18. Mar. 2025

Eindhoven university of technology
Department of mathematics and computer science

Master's thesis

Efficient Secure Computation of AES

Louiza Papachristodoulou

Eindhoven, July 2010

Supervisors:
Dr.ir. L.A.M. (Berry) Schoenmakers
Dr. J. (Jorge) Guajardo Merchan

Technische Universiteit Eindhoven University of Technology

PHILIPS sense and simplicity

Master's thesis

Efficient Secure Computation of AES

Eindhoven university of technology
Department of mathematics and computer science

Author:
Louiza Papachristodoulou

Supervisors:
Dr.ir. L.A.M. (Berry) Schoenmakers
Dr. J. (Jorge) Guajardo Merchan

Eindhoven, August 2010

Abstract

The main purpose of developing the Advanced Encryption Standard (AES) was to provide a standard algorithm for encryption, strong enough to secure U.S government documents for at least the next 20 years. The strong security of this algorithm raised the need to widely adopt it for a variety of encryption procedures, such as secure transactions via the Internet. A straightforward implementation of AES would be time-consuming and not convenient for commercial purposes. Therefore, a wide variety of approaches on efficient implementations of AES have appeared. Most of them seek to minimize the complexity of the circuit that performs the AES encryption, and more precisely, the computational workload of the S-Box of AES, which is the only non-linear part of the algorithm. Some implementations in this direction have already appeared (in VIFF and Fairplay platforms) and they make it feasible to use the AES algorithm in the secure multi-party setting. However, the S-Box of AES is still a bottleneck in the performance of such large circuits.

This thesis constitutes an extensive analysis of the most efficient approaches to implement the AES S-Box and how these can be used in a secure multiparty implementation of the AES algorithm. The AES algorithm is chosen as a case study and the techniques that are used are expected to be applicable more broadly. In particular, we describe how to implement a Yao circuit of the AES S-Box, which is significantly smaller than other alternatives in the literature. The straightforward binary circuit implementation of AES, even with the optimizations proposed by Pinkas et al., results in a large circuit with thousands of gates (33880 gates for the AES algorithm, with 8639 non-XOR gates that affect the computational workload of the circuit). The "tower fields" approach, proposed by Paar and applied to the AES circuit by Satoh et al., reduces significantly the number of required gates, resulting in small and compact circuits that can be used to implement the AES algorithm. Our proposed method takes advantage of the free-XOR technique of Kolesnikov as well to reduce the overall complexity of the circuit. Finally, this work concludes on the performance of the presented methods and on the efficiency of secure multiparty AES in practice.

Acknowledgements

I would like to thank the people who have been important to the accomplishment of my thesis.

Firstly, I would like to express my sincere gratitude to my supervisors Berry Schoenmakers and Jorge Guajardo Merchan both for involving me in this challenging research and the continuous support and enthusiasm shown. My special thanks to Berry for giving me the opportunity to conduct my research in cryptology and for his significant help in the Mathematica implementations. For his time devoted to help me to develop the ideas in my thesis and for the advices during my research, not only concerning my thesis, but also my future career life, I owe to thank Jorge. I would like to thank Jerry den Hartog for being in the exam committee and Henk van Tilborg, who first brought me in contact with the area of cryptology and kept giving me his useful advices even after the completion of his lecture.

Thanks to my colleagues at Philips Research for the pleasant working environment, the nice moments and discussions in and out of HTC, and in particular thanks to my office mates Maaike Schakenbos and Joeri de Ruiter for the cooperation, the exchange of ideas and for daily help and support.

I owe my lovely thanks to my family, these adorable three persons, who support me to every decision I make and are always next to me, even now that we live far away from each other. Last but not least, I am especially thankful to all my friends, in particular the ones that we have lived together in the same house and later in the same "box-block", because they successfully replaced this "family physical absence". Thank you for being next to me and for making these two years unforgettable. Special thanks to Chiara Tamiello and Marietta Gontikaki for taking care of me and to Despoina Mandilara, Glykeria Liontou, Semele Mylona, Döndü Sahin and Fotis Gonidis for their support and useful comments on my thesis.

Contents

1 Introduction **11**

1.1 Related Work 12

1.2 Scope of the Thesis 13

2 Mathematical Preliminaries **15**

2.1 Finite Field Arithmetic 15

2.2 Composite Fields 17

3 Secure Multiparty Computations **21**

3.1 Binary Circuit Approach 23

3.1.1 Definitions 23

3.1.2 Yao's Protocol 24

3.1.3 Improved Garbled Circuits 27

3.2 Arithmetic Circuit Approach - Secret Sharing 30

3.2.1 Shamir Secret Sharing 32

3.2.2 Verifiable Secret Sharing 33

3.2.3 Pseudorandom Secret Sharing 35

4 Secure Multiparty AES **37**

4.1 Advanced Encryption Standard (AES) 37

4.1.1 Algorithm Specifications 37

4.1.2 Implementation Issues 40

4.2 Multiparty Computation Protocols for AES 40

4.2.1 Secure Multiparty AES 41

4.2.1.1 Masked Exponentiation 41

4.2.2 Binary Circuit Approach - Secure Two-Party AES 42

4.2.2.1 Complexity 45
4.2.3 Straightforward methods of Constructing the AES S-Box . . . 45
4.2.3.1 Standard Implementation of AES S-Box 46
4.2.3.2 Improvement of the Standard Implementation using the Karatsuba Algorithm 49
4.2.4 Composite Field Implementation of AES S-Box 51
4.2.5 Composite Fields Approach with Normal Basis 58
4.2.5.1 A new combinational logic minimization technique . 63
4.2.6 Combinational Logic minimization with Free-XOR Technique 65
4.2.7 Comparison of the proposed Methods 69

Bibliography **73**

A Straightforward Method **77**

B Compact Inverter **81**

List of Figures

3.1 Half Adder 26
3.2 Adding a two-bit number 26

4.1 SubBytes transformation 38
4.2 Inverter over the composite field $GF(((2^2)^2)^2)$ 53
4.3 Squaring over the composite field $GF((2^2)^2)$ 54
4.4 Inverter over the composite field $GF((2^2)^2)$ 55
4.5 Multiplier over the composite field $GF((2^2)^2)$ 57
4.6 Inverter over the field $GF(2^8)$ using the normal basis 60
4.7 Normal multiplier over the field $GF(2^4)$ 61
4.8 The smallest Inverter over the field $GF(2^4)$ 64

Chapter 1

Introduction

The vast amount of digital data that people need to handle, store and share, usually consist of private information that should be secured against disclosure and misuse. As the need for data sharing and resource distribution increases, the problem of secure data sharing emerges. Secure information sharing should be preserved in modern infrastructures, for instance in *Cloud Computing*, where information, software and hardware resources are shared amongst the clients on demand. The technological infrastructure that supports the "cloud" and similar models, should provide the appropriate mechanisms that preserve the security and privacy of the corresponding data.

Encryption of the stored data using the traditional encryption algorithms is not suitable for this setting, since only one entity would know the private key and nothing can be shared with other participants, unless each one shares his private key. In the above mentioned cases, the concept of *Secure Multiparty Computation*(MPC) seems the most appropriate practical way to preserve privacy. The main problem in secure MPC is to find a protocol for the participants $P_1, ..., P_l$, which enables them to jointly compute the output value $f(x_1, ..., x_l)$ of a computable function f without leaking any information on their respective secret input values $x_1, ..., x_l$. The only information that could be leaked is the one inferred logically from the output value.

In terms of encryption algorithms, the concept of MPC can be applied for the computation of the encryption function $E(\cdot)$, which transforms the initial message m to the ciphertext c, that is $E(m) = c$. By using MPC, each participant $P_i, i = \{1, ..., l\}$ distributes shares of its private input x_i, where x_i can be a share of the message m or the private key K, among the other parties, and thus makes x_i available in a shared form. Then, all the participants (or a qualified majority of them in the case of threshold systems) could combine their shares to evaluate the circuit corresponding to the function $E(\cdot)$. Each gate of this circuit is evaluated one by one,

ensuring the property that if all the inputs to a gate are available in a shared form, then then output of the gate is also produced in a shared form [Sch05]. By reconstruction of the output, evaluation of the final gate of the circuit for E will produce the output value $c = E(m_1, ...m_l)$ in shared form. By applying the reconstruction protocol of the secret sharing scheme, the value of c is obtained. The participants in this protocol can not derive any information apart from their initial shares and the final ciphertext, which cannot be decrypted unless they cooperate to retrieve the secret key K. Moreover, each party of the protocol acts as dealer and as participant during the execution of the protocol, and there is no need to trust a third party, in order to evaluate the final output of the encryption. The trusted third party can be "emulated" by the mutually distrustful parties themselves.

The honest participants of this type of protocols want to preserve some security properties, such as privacy and correctness. Many protocols presented in the field of MPC provide security against passive or active adversaries by using techniques of *secret sharing* or *garbled circuits*. The most important protocols will be analyzed in this thesis, focusing mostly on their efficiency in practical implementations of MPC.

1.1 Related Work

The first general solution to the problem of secure two-party computation against semi-honest adversaries was introduced by [Yao86] and it is based on garbled circuits. Despite the fact that MPC has been considered by the cryptographic community for many years, the complexity for constructing circuits that represent this function used to be inefficient. Therefore, not many practical implementations of MPC were presented. In general, MPC protocols are categorized according to the way the function f is represented, as the *binary circuit approach* representing the function as a binary circuit, and the second approach, which is based on secret sharing and/or homomorphic encryption and operates on *arithmetic circuit* representation of the computed function.

Yao's protocol follows the binary circuit approach and it is remarkably efficient, since it has a constant number of rounds and uses one oblivious transfer per bit only. A complete proof for Yao's protocol was presented in [LP04] and the first efficient computations of secure two-party problems were performed in Fairplay platform by Malkhi et al. [MNPS04]. Some years later, Pinkas et al. showed that secure two-party computations can be implemented in practice, even for large circuits, such as the ones performing AES encryption [PSSW09].

Secure MPC protocols using an arithmetic circuit construction over a suitable field

are usually related to a secret sharing scheme. For the implementation of MPC protocols following this approach, the software framework VIFF was firstly released in 2004 and it was later used to implement MPC protocols for real-life projects, such as SIMAP for secure auction without the participation of a trusted party. More details about this research project were reported in [BCD+08]. Depending on the function to be computed, the implementation complexity for computing large circuits can be reduced with this approach, since it exploits the arithmetic properties of the functions that need to be computed. A secure multiparty version of the AES algorithm was developed also for VIFF by Marcel Keller. Using the `viff.aes` module, it is possible to securely compute a secret shared AES encrypted ciphertext of a (possibly) secret shared plaintext with a (possibly) secret shared key. The inputs have to be given either as a list of shares over GF256 (byte-wise) or as a string. The runtime has to be able to handle shares over GF256. The arithmetic properties of AES makes its computation by arithmetic circuits relatively fast. For instance, encrypting a 128-bit block using a 128-bit secret shared AES key takes about 2 seconds using three machines. Decryption is not implemented yet.

At the same time, a wide variety of approaches to implementing AES and minimizing its circuitry have appeared independently of the MPC setting. At first, Rijmen [Rij01] suggested to use the subfield arithmetic in the crucial step of computing the multiplicative inverse in the Galois field $GF(2^8)$, i.e. reducing an 8-bit calculation to several 4-bit ones. Satoh et al. in [SMTM01] further extended this idea by using the "tower field" approach and breaking-up the 4-bit calculations into 2-bit ones; the most significant reduction in the number of multiplications needed to calculate the inverse in $GF(((2^2)^2)^2)$ was achieved in this way. Canright [Can05], who reduced further the total number of required operators and chose to calculate the inverse by using the normal basis representation of the fields, inspired Boyar and Peralta to construct the most compact AES S-Box implementation up-to-date [BP10].

It seems very interesting to apply some ideas of the above mentioned methods for compact hardware architectures of the AES algorithm to the MPC setting and therefore, to create practical and faster ways to implement secure multiparty AES.

1.2 Scope of the Thesis

The research interest on protocols for secure MPC has arisen, due to the software frameworks that have been recently presented and make it possible to perform secure MPC in practice. Such protocols can be useful for many applications, such as secure auctions, cloud computing, benchmarking and in general, preserving the privacy of the users in large, distributed modern systems.

This thesis describes the different approaches for implementing secure MPC and compares the efficiency of some implementation systems on large circuits. For this purpose, AES algorithm was chosen as an example application of these implementations, since it is based on arithmetic in $GF(2^8)$ and the computation workload of its operations, such as the S-Box, is quite large. Based on the "tower field" approach and the work of Satoh et al. [SMTM01], we analyzed the construction of one of the smallest up-to-date AES inversion circuit. This method of splitting the $8-bit$ operations into $4-bit$ ones and then into 2−bit calculations can be applied in digital circuits in any hardware implementation of this kind.

An extensive analysis of the most efficient approaches to implement the AES S-Box and how these can be used in a secure multiparty implementation of the AES algorithm are presented. More precisely, this thesis is organized as follows: in *Chapter 2* some mathematical preliminaries about finite fields and composite fields are presented, in order to make it possible for the reader to understand the "tower fields" approach, which offers the most efficient implementation of AES S-Box so far. In *Chapter 3*, the basic notions of secure multiparty computations (MPC) are presented. MPC comes essentially in two flavors; the arithmetic circuit representation, which is based on secret sharing and is more efficient in representing addition and multiplication operations, and the binary circuit approach, which handles binary operations, such as comparisons, most efficiently. The advantages and disadvantages of each approach and some useful implementation details are contained in this chapter. *Chapter 4* includes the main analysis of efficient secure AES implementations. After a brief description of the AES algorithm, the most efficient implementations of the two different approaches (arithmetic and binary) are presented. The orientation of the research focuses more on the binary circuit approach. In particular, we describe how to implement a Yao circuit of the AES S-box, which is significantly smaller than other alternatives in the literature. Finally, the number of elementary operators for the construction of an inversion circuit for AES are counted and the research focused on reducing the number of AND gates (multiplicative complexity) of this circuit by applying techniques developed by Canright, Boyar and Peralta. The construction of the AES S-Box in the polynomial basis representation of Satoh et al. is verified using Mathematica.

Chapter 2

Mathematical Preliminaries

Number theory and finite field theory are two mathematical areas that play a crucial role in cryptography. In our case, Rijndael's finite field $GF(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ is going to be extensively used. Therefore, this chapter focuses on operations in this field.

It is assumed that the reader is already familiar with elementary number theory and with the basic notions of *field theory*. However, composite field arithmetic and tower fields are not commonly taught at master's level. For the comprehension of the techniques applied in the next chapters, a brief introduction to the theory of finite fields is given in the next section, followed by a more detailed description of the theory of composite fields. At some points, where some theorems of number theory are necessary, they are going to be written down explicitly.

2.1 Finite Field Arithmetic

In most of the number systems used in elementary arithmetic, there are two binary operators that define the relations between the system's elements: addition and multiplication. A finite field is a type of algebraic structure that shares some of the basic properties of the commonly known number systems, as it is essentially a mapping from the integers to a finite subset of them.

Definition 2.1.1 *Let F_p be the non-empty set of integers $\{0, 1, ..., p-1\}$, where p is a prime number. Let $\phi : \mathbb{Z}/(p) \rightarrow \mathbb{F}_p$ be a one-to-one and onto mapping from the ring of residue classes of the integers modulo the prime p to the set $\mathbb{F}_p$, defined by $\phi([\alpha]) = \alpha$ for $\alpha = 0, 1, ..., p-1$. Then $\mathbb{F}_p$ fitted with the field structure induced by ϕ is a finite field, called the* ***Galois Field*** *of order p. An alternative notation of the finite field $\mathbb{F}_p$ is $GF(p)$.*

A **Galois Fields** is a field that contains only finitely many elements. The mapping $\phi : \mathbb{Z}/(p) \rightarrow \mathbb{F}_{|}$ is an isomorphism, which means that $\phi([\alpha] + [\beta]) = \phi([\alpha]) + \phi([\beta])$ and $\phi([\alpha] \cdot [\beta]) = \phi([\alpha]) \cdot \phi([\beta])$. The finite field $\mathbb{F}_p$ has zero element 0, identity element 1 and its structure is exactly the structure of $\mathbb{Z}/(p)$. All the operations on elements over finite fields are performed modulo the order of the field p and produce a result that belongs also to the same field.

It is a well-known fact, that for p prime, the field $\mathbb{F}_p$ is the unique finite field with p elements. The prime number p is called the characteristic of the field. For any positive integer n, it is possible to construct the finite field $\mathbb{F}_{p[x]} = GF(p^n)$ with p^n elements, which can be represented as polynomials $f \in \mathbb{F}_{p[x]}$ of degree less than n over $GF(p)$. Operations can be performed modulo an irreducible polynomial $q(x)$ with $deg(q(x)) = n$ over $GF(p)$. The residue class ring $F[x]/(f)$ is a ring if the following theorem holds:

Theorem 2.1.1 *For $f \in F[x]$, the residue class ring $F[x]/(f)$ is a field if and only if f is irreducible over F.*

An important remark for finite fields of characteristic 2, is that they can be expressed as binary numbers, with the coefficients of each term in the polynomial representing the one bit in the corresponding binary expression of the element. For a hardware representation, addition of two elements in $GF(2)$ is *exclusive OR (XOR)* and multiplication is an *AND* gate, as it can be seen in the following tables:

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1: AND-gate

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.2: XOR-gate

The addition of two elements in a field of characteristic 2, $GF(2^n)$, is equivalent to n-XOR gates. However, the multiplication of n-bit numbers is a more complicated operation, since it involves a combination of AND and XOR gates. The above mentioned table for the AND gate can be applied when the components of the multiplied elements are analyzed in 2-bit elements of a lower field. As is mentioned in Chapter 1, the AES encryption algorithm is going to be the main application of the implementation issues that are going to be discussed in the next sessions. The AES specification is going to be described in Chapter 4. However, at this point is interesting to analyze the arithmetic properties of the *Rijndael* field $GF(2^8)$ with irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$, which is used in the *SubBytes* transformation. The operation that provides the non-linearity of the cipher is the

multiplicative inverse of each byte over $GF(2^8)$, which increases dramatically the computational complexity of the algorithm.

In general, inversion of a polynomial $f(x)$ over finite fields can be performed using the extended Euclidean Algorithm for polynomials. Given an irreducible polynomial $p(x)$ and a polynomial $f(x)$, the inverse of $f(x)$ can be found as follows:

$$ap(x) + bf(x) = 1 = gcd(f,p) \Rightarrow bf(x) \equiv 1 \quad modp(x) \Rightarrow b \equiv f(x)^{-1} \quad modp(x)$$

For finite fields of order q, where $q = p^n$, Fermat's Little Theorem can be applied to find the multiplicative inverse as usual.

Theorem 2.1.2 *Let $\mathbb{F}_q$ be a finite field with $q = p^n$ elements and $p(x)$ an irreducible polynomial of degree n over $\mathbb{F}_p$. Then for every element $\alpha \in \mathbb{F}_q$ it holds:*

$$a^q = a \quad \Leftrightarrow \quad a^q \equiv a \quad mod\, p(x).$$

For the finite field $GF(2^8)$, the multiplicative inverse of an element a will be a^{254}:

$$a^{2^8} \equiv a\, mod\, p(x) \Rightarrow a^{255} \equiv 1\, mod\, p(x) \Rightarrow a^{254} \cdot a \equiv 1\, mod\, p(x) \tag{2.1}$$

2.2 Composite Fields

According to [LN03], the following definitions regarding field extensions can be given:

Definition 2.2.1 *Let F be a field and K a subset of F. The subset K is itself a field under the operations of F and it is called a subfield of F. In this context, F is called an extension of K.*

If L is an extension field of K, then L can be viewed as a vector space over K. All the elements of L form an abelian group under addition, each "vector" of L can be multiplied by element $r \in K$ and the result ra is again in L.

Definition 2.2.2 *Let L be an extension field of K. If L, considered as a vector space over K is finite-dimensional, then L is called a finite extension of K. The dimension of the vector space L over K is then called the degree of L over K, which is a positive integer and it is denoted as $[L : K]$.*

It is possible to derive the same field by using different degree extensions, as long as the number of field elements are the same. The derived fields will have the same algebraic structure and only the representation of the field elements is going to differ. Actually, the following theorem for the existence and uniqueness of finite fields holds:

Theorem 2.2.1 *For every prime p and every positive integer n there exists a unique finite field with p^n elements. Any other finite field with $q = p^n$ elements is isomorphic to the splitting field of $x^q - x$ over $\mathbb{F}_p$.*

A complete proof is given in [LN03]. This theorem essentially shows that there exists only a unique field of a specific order and justifies the fact that the term "*the* Galois field of order q" is used. All the other fields with the same order q are isomorphic to *the* Galois Field $GF(q)$ and they have the same number of elements q.

In the following chapters, we are going to deal with finite field extensions of $GF(2)$ and more precisely with the Galois Field $GF(2^8)$. From the previous theorem, it is obvious that we can derive the same field, if we apply multiple degree-2 extensions to the field $GF(2)$ under a polynomial basis using the appropriate irreducible polynomials. Actually, the following fields are isomorphic:

$$GF(2^8) \cong GF((2^4)^2) \cong GF(((2^2)^2)^2) \tag{2.2}$$

In [SMTM01], the composite field $GF(((2^2)^2)^2)$ is used for an efficient implementation of the AES S-Box. This field is constructed by repeating degree-2 extensions to the field $GF(2)$ under a polynomial basis using the following irreducible polynomials:

- $GF(2^2)$: $x^2 + x + 1$
- $GF((2^2)^2)$: $y^2 + y + x$
- $GF(((2^2)^2)^2)$: $z^2 + z + (x + 1)y$

The main reason for using this field representation is that by reducing the calculations from 8−bit to several 2−bit ones, the complexity of the computations is also reduced. A detailed description of the construction of the composite field is given now.

- Galois field $GF(2^2)$ with irreducible polynomial $x^2 + x + 1$.
  The elements of this field are polynomials of degree less or equal than 1 with coefficients in $GF(2) = \{0, 1\}$. Therefore, the field consists of the following elements:

  $$GF(2^2) = \{\alpha \times x + \beta | \alpha, \beta \in GF(2)\} = \{0, 1, x, x + 1\}. \tag{2.3}$$

  Note that the inverse of these elements equal to the squaring of the element, apart from 0 that is mapped to 0. Indeed, $1^{-1} = 1^2 = 1$, $x(x + 1) = x^2 + x = 1, x^2 = x + 1$, so $x^{-1} = x + 1$ and the same holds for $(x + 1)^{-1}$, since $(x + 1)^2 = x^2 + 1 = x = x^{-1}$.
  Thus, inversion is a linear operation in $GF(2^2)$, since squaring is linear over any field of characteristic 2.

- Galois field $GF((2^2)^2)$ with irreducible polynomial $y^2+y+\phi$, where $\phi = \{10\}_2$. The elements of $GF((2^2)^2)$ are polynomials of degree less or equal than 1 with coefficients in the field $GF(2^2)$. The irreducible polynomial is $y^2 + y + x$. Taking y as the variable in this case, the elements can be written down as follows:

$$GF((2^2)^2) = \{\alpha \times y + \beta | \alpha, \beta \in GF(2^2)\} = \{0, 1, y, y+1, x, x+1, x+y, x+y+1, xy, xy+1, xy+x, xy+(x+1), (x+1)y, (x+1)y+1, (x+1)y+x, (x+1)y+(x+1)\}$$

- Galois field $GF(((2^2)^2)^2)$ with irreducible polynomial $z^2 + z + \lambda$, where $\lambda = \{1100\}_2$.
The field $GF(((2^2)^2)^2)$ consists of 256 elements, which are polynomials of degree less or equal than 1 with coefficients in $GF((2^2)^2)$. According to the previous definitions, the irreducible polynomial in this field is $z^2+z+(x+1)y$. Hereby, some elements of $GF(((2^2)^2)^2)$ are listed:

$$GF(((2^2)^2)^2) = \{\alpha \times z + \beta | \alpha, \beta \in GF((2^2)^2)\} = \{0, 1, y, y+1, x, x+1, x+y, x+y+1, xy, xy+1, (x+1)y+x+1, z, z+1, z+y, z+xy, yz, yz+1, (y+1)z, (y+1)z+1, (y+x)z+1, xyz, xyz+1, (x+1)yz, x(y+1)z+1, ...\}$$

In the following chapters, we are going to use different basis representations to compute the multiplicative inverse of elements over finite fields. Therefore, it is necessary at this point to remind some definitions of basic characteristics of finite fields. Regarding a finite field extension $F = \mathbb{F}_{q^m}$ of the finite field $K = \mathbb{F}_q$ as a vector space over K, and if $\{a_1, ..., a_m\}$ is a basis of F over K, each element $a \in F$ can be uniquely represented in the form:

$$a = c_1 a_1 + ... + c_m a_m \quad with \quad c_j \in K \quad \forall 1 \leq j \leq m$$

Definition 2.2.3 *Let $\mathbb{F}_{q^m}$ be an extension of $\mathbb{F}_q$ and let $\alpha \in \mathbb{F}_{q^m}$. Then the elements $a, a^q, ..., a^{q^{m-1}}$ are called the conjugates of a with respect to $\mathbb{F}_q$.*

If a is a root of an irreducible m-degree polynomial f in $\mathbb{F}_q$, then the conjugates of a are also roots of f. The following example makes this definition clear:

Example 2.2.1 *Let $a \in \mathbb{F}_{16}$ be a root of $f(x) = x^4 + x + 1 \in \mathbb{F}_2[x]$. Then the conjugates of a with respect to $\mathbb{F}_2$ are $a, a^2, a^4 = a+1, a^8 = a^2+1$ and they are also primitive elements of $\mathbb{F}_{16}$. The conjugates of a with respect to $\mathbb{F}_4$ are $a, a^4 = a+1$.*

The trace from F to K is a linear mapping that is defined as follows:

Definition 2.2.4 *For $a \in F = \mathbb{F}_q$ and $K = \mathbb{F}_q$, the trace $Tr_{F/K}(a)$ of a over K is defined by*

$$Tr_{F/K}(a) = a + a^2 + ... + a^{q^{m-1}} \tag{2.4}$$

Simply said, the trace of an element over a finite field is the sum of the conjugates of this element.

Another interesting function from a finite field to a subfield is the norm, which is obtained by forming the product of the conjugates of an element of the field with respect to the subfield.

Definition 2.2.5 *For $a \in F = \mathbb{F}_q$ and $K = \mathbb{F}_q$, the norm of $N_{F/K}(a)$ of a over K is defined as*

$$N_{F/K}(a) = a \cdot a^q \cdot ... \cdot a^{q^{m-1}} = a^{(q^m-1)(q-1)} \tag{2.5}$$

Both the trace and the norm are field elements, since they are obtained from addition and multiplication of field elements respectively.

The two types of bases that will be used in *Chapter 4* are the polynomial and the normal bases, which are defined as follows:

Definition 2.2.6 *Let a be a primitive element of $\mathbb{F}_{q^m}$. Then, the basis $\{1, a, ..., a^{m-1}\}$ is called the polynomial basis of the field.*

Definition 2.2.7 *Let $K = \mathbb{F}_q$ and $F = \mathbb{F}_{q^m}$. Then a basis of F over K of the form $\{a, a^q, ...a^{q^{m-1}}\}$, consisting of a suitable element $a \in F$ and its conjugates with respect to K, is called a normal basis of F over K.*

As it is mentioned earlier, the AES algorithm uses the particular Galois field of 8-bits $GF(2^8)$, where the bits are coefficients of a polynomial and multiplication is modulo the irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$, with addition of coefficients modulo 2. Let A be one root of $q(x)$; then the polynomial basis is $[A^7, A^6, A^5, A^4, A^3, A^2, A, 1]$ and an element of $GF(2^8)$ can be expressed as a polynomial of degree at most 7 with coefficients in $GF(2)$.

Alternatively, we can represent an element of $GF(2^8)$ as a polynomial in z with coefficients in $GF(2^4)$ using the normal basis $[Z, Z^{2^4}]$, where Z, Z^{16} are the roots of the irreducible polynomial $r(z) = z^2 + Tz + N = (z + Z)(z + Z^{16})$. T and N is the trace and norm of the element Z respectively.

Chapter 3

Secure Multiparty Computations

As it is mentioned in Chapter 1, the aim of this thesis is to analyze the concept of secure multiparty computation in practice, focusing on applications to the AES algorithm. The main problem in secure multiparty computations is to find a protocol for the participants $P_1, ..., P_l$, which enables them to jointly compute the output value $f(x_1, ..., x_l)$ of a computable function f without leaking any information on their respective secret input values $x_1, ..., x_l$. The only information that could be leaked is the one inferred logically from the output value. The concept of secret sharing plays a crucial role in secure multiparty computations, therefore, we hereby provide the problem statement:

Problem Statement: *Let $[x]$ be a value that should be secretly shared among parties $P_1, P_2, ..., P_l$. For each $i \in \{1, ..., l\}$, the party P_i should hold the secret shared value x_i and all the parties want to jointly evaluate the outcome of the function $f(x_1, ..., x_l)$ for some given computable function f. The problem of secure multiparty computation is to find a protocol for $P_1, ..., P_l$, which enables them to jointly compute the output value $f(x_1, ..., x_l)$ without leaking any information on their respective secret input values $x_1, ..., x_l$, except from the information that can be inferred logically from the output value. It is proven that a protocol for evaluating a given function f securely can always be found, when f is a computable function.*

In general, these protocols can be categorized according to the way the function f is represented, as follows:

- The *binary circuit approach* represents the function as a binary circuit and it is particularly useful when binary operations, such as comparisons, need to be computed. It is also very efficient when only two parties are involved. However, this approach handles arithmetic operations, especially multiplications, less efficiently.

- The second approach is based on secret sharing and operates on *arithmetic circuit* representation of the computed function. This approach is usually applied when there is an honest majority among the participants and is better at representing addition and multiplication operations. It also guarantees information theoretic security of the inputs and outputs of the computations.

In 1986, Yao presented a constant-round protocol for secure two-party computations in the semi-honest model. That was the first most efficient method to evaluate the function f using boolean circuits. In this model the two participants follow the protocol exactly, but they try to retrieve more information that they are supposed to learn by analyzing the transmitted messages during the execution of the protocol [Yao86]. A formal proof of security of Yao's protocol was given by Lindell and Pinkas in [LP04] in an attempt to clarify and justify the correctness of the most basic result in the field of secure multiparty computations. Some years later, the same authors presented an efficient protocol for secure two-party computations in the presence of malicious adversaries [LP07]. Yao's result is of great importance for this field, because it was the first general solution to the two-party problem and most of the later results and extensions of this construction are based on his protocol. During the last years, many protocols for secure multiparty computation have appeared, but their implementations are quite restricted to their application.

A *Secure Two-Party Computation System* that implements of Yao's protocol in practice is **Fairplay**. Fairplay is a full-fledged system that implements generic secure function evaluation in real settings. In [MNPS04] the system and it's functionality are described. This system was a first implementation of a two-party computation using boolean circuits. For *secure multiparty computations* the **VIFF** (Virtual Ideal Functionality Framework) was introduced in 2007. VIFF allows the execution of secure multi-party computations, in which a number of parties (three or more at the moment) execute a cryptographic protocol to do some joint computation without revealing their input values. The protocols and circuit constructions that are going to be discussed in the section for Arithmetic Circuit Approach, can be constructed and implemented in the VIFF platform.

This chapter, after some necessary definitions, describes both the "binary circuit" and "arithmetic circuit" approach on how to efficiently perform secure multiparty computations. A description of Yao's protocol and the Free-XOR proposal of Kolesnikov and Schneider for improving the garbled circuits are presented, since these are the most important results in the binary approach. Moreover, an example of how a *half-adder* works in terms of garbled circuits is given, in order to provide the reader with a clear understanding of the main construction for garbled circuits. Then, the arithmetic circuit approach is presented by analyzing some Secret Sharing techniques.

3.1 Binary Circuit Approach

3.1.1 Definitions

In this section, some definitions are given, which are going to be used throughout this chapter. We start with the definition of the *random oracle model*, which is a useful abstraction introduced and justified by [BR93] and was used to prove the security of the algorithm designed to construct garbled circuits.

Definition 3.1.1 ***Random Oracle*** *is a mathematical function* $\{0,1\}^* \mapsto \{0,1\}^N$ *that maps every possible query to a random response chosen uniformly from its output domain. For any specific query it responds in the same way every time the oracle receives this query.*

In practice, random oracles are implemented by hash functions, such as SHA-1.

An important concept in secure multiparty computations is *secure function evaluation.*

Definition 3.1.2 *A two-party* ***Secure Function Evaluation*** *(SFE) is a procedure, which allows two parties to securely evaluate a function on their respective inputs* x, y*, while privacy of both inputs is maintained.*

The construction of efficient SFE algorithms can enable the conduct of a variety of electronic transactions even if there is mutual mistrust of the participants.

Oblivious Transfer is a fundamental primitive of secure computations. It is a protocol by which a sender sends some information to the receiver, but it remains oblivious to the sender what is received. The first form of oblivious transfer was introduced by [Rab81] and it is based on the RSA cryptosystem. In this form, the sender sends a message to the receiver with probability 1/2 and he remains oblivious to whether or not the receiver received the message. Goldreich et al. proposed a more practical form of oblivious transfer called *1-out-of-2 Oblivious Transfer* that could be used to build protocols for secure multiparty computations [EGL85]. This approach is defined as follows:

Definition 3.1.3 *The* ***1-out-of-2 Oblivious Transfer*** *(OT) is a two-party protocol that works as follows:*

- *The sender* P_1 *generates two secrets* m_0, m_1*.*
- *The receiver* P_2 *selects a bit* $i \in \{0,1\}$*, which corresponds to the secret that he requests from the sender.*

- *At the end of the protocol, P_2 learns the secret m_{i_c}, where i_c corresponds to the bit of his choice, but nothing about m_{1-i_c} and P_1 learns nothing about i.*

3.1.2 Yao's Protocol

The concept of two party computation involves two parties A and B with respective private inputs x and y respectively, who wish to jointly compute a function $f(x, y) = (f_1(x, y), f_2(x, y))$, such that the first party receives $f_1(x, y)$ and the second party receives $f_2(x, y)$. This function can represent a random variable or a relation $\mathcal{R} = \{<, >, =, \leq, \geq\}$ between the inputs of the two parties or a more complicated relation. The security requirements of a *secure two party computation* are that nothing can be revealed from the protocol apart from the output (*privacy*) and that the output is distributed to both parties in the same way covering the specified functionality (*correctness*). The development of two-party *SFE algorithms* allows the performance of electronic transactions between mutually mistrusted participants.

Yao's protocol for secure two party computations in the semi-honest model achieved high levels of efficiency, since it has a constant number of rounds (independent from the number of inputs or the size of the circuit), it uses one oblivious transfer per input bit only and no additional oblivious transfers are needed for the rest of the computations. We are going to use Yao's protocol for evaluating the SFE of private functions (PF-SFE), whereby one party constructs the function and the other one needs to evaluate the output of the function according to his private input. Assuming that *Party A* is the constructor and *Party B* is the evaluator, Yao's protocol can be described in the following steps:

1. The computed function $f(x, y)$ should be converted into a boolean circuit, which can perform the same operations with the function. The constructor of the circuit, party A, knows the topology of the circuits and the type of the gates. Therefore, he is the only participant that can construct the corresponding function. Party A evaluates the output of one gate and then generalizes the procedure to evaluate the whole circuit securely, in order to verify the correctness of the circuit.

2. Party A picks two random keys for each wire, so that one corresponds to input value 1 and the other corresponds to 0. Accordingly, two keys should be chosen for the output wire, one for each possible output. In this way, six keys are created for a gate with 2-input wires

3. For each gate, Party A encrypts each row of the truth table and produces an Encrypted Truth Table. The encryption method uses the pair of input-wire keys to encrypt the corresponding output-wire key. In this way, each gate is replaced by a 4-entry table indexed by the values of the keys used for the encryption.

4. Party A randomly permutes (garbles) the Encrypted Truth Table and sends it to Party B. In this way, party B does not know which row of the garbled table corresponds to the original table.

5. Party A sends the key corresponding to his input bit. For instance, if A's input is 1, he sends k_{1x} to B, otherwise he sends k_{0x}. Party B learns $k_{b'x}$, where b' is A's input bit, but he cannot reveal the real b', since the chosen keys are random and they do not represent the real values, they just correspond to them. If B was able to reveal the true value of b', then he could find in which row of the original table the encryption started, and by using his input key-bit, he could learn the output value, which represents the encrypted output key.

6. Both parties run the Oblivious Transfer protocol, in order to exchange the key corresponding to B's input bit. Party A has generated the keys for all the wires. Therefore, he uses the two keys corresponding to B's wire as input to the protocol. B chooses one bit and then he learns only the key that corresponds to the input bit of his choice. At the end, B knows $k_{b'x}$ and k_{bx}, where b is the bit of his choice. With these keys, he can calculate exactly one output, the one that corresponds to the output-wire key that is encrypted with the specific b and b'.

7. For each wire in the circuit, party B learns only one key that corresponds to 0 or 1, but he does not know which value it actually represents. So according to his input, he can evaluate the final outcome of the circuit, without learning the intermediate values. In this way, Yao's protocol prevents party B from learning the type of gates present in the circuit. Party B only knows the topology of the circuit and this reveals nothing about the input of party A. Furthermore, party B sends the key for the final output to party A and tells him if it corresponds to 0 or 1. However, party B does not reveal the intermediate wire keys to party A, because party A knows the type of each gate, and he therefore can understand by the output of each wire, what the real value of each key is.

Example 3.1.1 *Assume that two parties A and B have two numbers and they want to calculate the carry after adding them, in order to use it as input in another circuit or application. However, they want to perform the addition operation without revealing to each other what their private inputs are. With* $A, A', B, B' \in \{0, 1\}$*, we denote the binary representation of these two-bit numbers respectively. The addition function is easily converted to a boolean circuit, by using full- or half-adders. In this case, two full-adders should be used, one for each digit of the numbers. Each of these contains two half-adders.*

The half-adder and its original truth table are depicted in Figure 3.1.

Figure 3.2 shows the two full-adders that are needed for adding two-bit numbers.

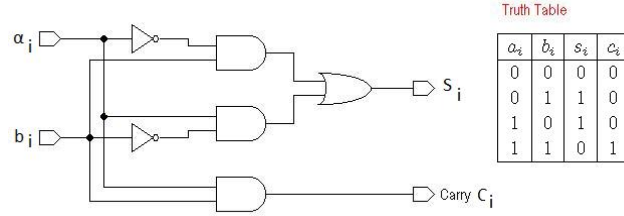

Figure 3.1: Half Adder

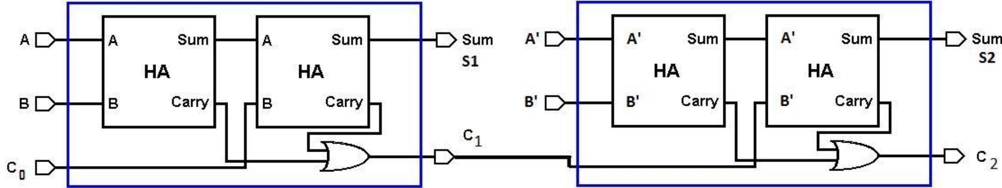

Figure 3.2: Adding a two-bit number

Both circuits include an AND and a XOR gate. The output of the XOR gate is the sum s_i of the two bits and the output of the AND gate is the carry c_i, $i \in \{0, 1\}$. For the first circuit, we assume that $c_0 = 0$.

If A, B are the input values for the first circuit, and A', B' are input values for the second circuit, then the original truth tables for both circuits will be similar to the ones presented above, with $a_0 = A, a_1 = A', b_0 = B, b_1 = B'$.
Party A has to pick two random keys for each wire, one corresponding to input value 1 and the other to input value 0. Following the steps of Yao's protocol, we have to calculate the encrypted truth table for each pair of input values. With k_{a0}, k_{a1} we denote the keys corresponding to values 0 or 1 for the wire a. Similar notation is going to be used for the other wires and the encrypted table will be the following:
Then, party A sends a permutated table to party B after garbling the rows in the

a_i	b_i	s_i	c_i
k_{a0}	k_{b0}	$E_{ka0}(E_{kb0}(s_0))$	$E_{ka0}(E_{kb0}(c_0))$
k_{a0}	k_{b1}	$E_{ka0}(E_{kb1}(s_1))$	$E_{ka0}(E_{kb1}(c_0))$
k_{a1}	k_{b0}	$E_{ka1}(E_{kb0}(s_1))$	$E_{ka1}(E_{kb0}(c_0))$
k_{a1}	k_{b1}	$E_{ka1}(E_{kb1}(s_0))$	$E_{ka1}(E_{kb1}(c_1))$

Table 3.1: Encrypted Truth Table

encrypted table. The garbled encrypted table will have the form of Table 3.2.
For this example, assume that party A has the string 10 *and party B the string* 01*, then party A has to send the key k_{a0} corresponding to his input bit 0. Both parties*

a_i	b_i	s_i	c_i
k_{a1}	k_{b1}	$E_{ka1}(E_{kb1}(s))$	$E_{ka1}(E_{kb1}(c))$
k_{a1}	k_{b0}	$E_{ka1}(E_{kb0}(s))$	$E_{ka1}(E_{kb0}(c))$
k_{a0}	k_{b0}	$E_{ka0}(E_{kb0}(s))$	$E_{ka0}(E_{kb0}(c))$
k_{a0}	k_{b1}	$E_{ka0}(E_{kb1}(s))$	$E_{ka0}(E_{kb1}(c))$

Table 3.2: Garbled Encrypted Table

start the oblivious transfer protocol, in order to exchange B's input key securely. Since the first bit that B will use for the addition is 1*, he should choose the value* 1 *for the oblivious transfer, and then he is going to obtain the key* k_{b1} *corresponding to 1. Due to the oblivious transfer protocol, party A cannot find out which bit is chosen by the other party. Similarly, the keys corresponding to the other input bits are exchanged for the second circuit. According to his input, party B can evaluate the final output of the circuit by decrypting the output value* $E_{ka1}(E_{kb0}(c))$*. Then, he sends the key* k_{b0} *for the final output to A and tells him that it corresponds to 0, in order to make it possible for A to decrypt it.*

In general, the entries in the encrypted tables, i.e. the values that are encrypted with the keys $k_{a0}, k_{a1}, k_{b0}, k_{b1}$, are not the sum s_i and the carry c_i, but they are encrypted data or keys corresponding to s_i, c_i and which can be used as input to other gates of the circuit. In this way, the evaluator of the circuit cannot learn the intermediate input values of the other party. In this example, we created for simplicity the encrypted truth table for only one gate and we used the values of s_i, c_i in their plain form, in order to indicate which values are encrypted with which keys.

3.1.3 Improved Garbled Circuits

In [Kol05], a new garbled circuit construction for two-party secure function evaluation in the semi-honest model is presented. The most significant result of this construction is that XOR gates can be evaluated for free, which means without the use of associated garbled tables and the corresponding hashing or symmetric key operations. As it is mentioned in section 3.1.2, garbled tables are used to evaluate the output wire value of a garbled circuit, without revealing any information on the corresponding inputs. In [Kol05] construction, garbled tables should be used for the evaluation of all the other gates, apart from XOR and $NOT-$gates. The entries of these tables are encrypted garblings of the output wires. The garblings of the input wires serve as keys to decrypt the "right" output values.
The main assumptions that this protocol relies on are the following:

1. Acyclic boolean circuits with k-gates and arbitrary fan-out.

The single output of each gate can be used as input to an arbitrary number of gates.

2. Gates are topologically ordered.
   Let $G_1, ..., G_k$ be the gates of the circuit. When the gates are ordered in this way, then it holds that the input of the $i-$th gate G_i has no inputs that are outputs of a successive gate G_j, where $j > i$.

3. Semi-honest model.
   In the semi-honest case, the participants follow the protocol, but they try to retrieve more information from the execution of the transcripts.

Based on these assumptions, Kolesnikov constructed XOR gates that have a restrictive global relationship on the wire secrets, while at the same time preserve the security of the circuit and can be evaluated for free. Each XOR-gate with $n > 2$ is replaced with $n - 1$ two-input XOR-gates. The NOT gates can also implemented for free by eliminating them and inverting the correspondence of the values and garblings in the wires. All the other gates are implemented using standard garbled tables. More precisely, each gate with n-inputs is assigned to a table with 2^n randomly permuted entries.

A garbled circuit in [Kol05] is constructed according to the following algorithm:

1. Choose a random global key offset $R \in \{0,1\}^N$.
   This randomly chosen constant R is used for garbling the two values of each wire.

2. For each input wire W_i of the circuit C, each garbling $w = \langle k, p \rangle$ consists of a key $k \in \{0,1\}^N$ and a permutation bit $p \in \{0,1\}$. The garbled values of the wires should be set as follows:

$$w_i^0 = \langle k_i^0, p_i^0 \rangle \in_R \{0,1\}^{N+1} \quad (3.1)$$
$$w_i^1 = \langle k_i^1, p_i^1 \rangle = \langle k_i^0 \oplus p_i^0 \oplus 1 \rangle \quad (3.2)$$

   The first garbled value w_i^0 is randomly chosen, while the second one is related to the first by XORing each component with the random global offset R.

3. For each gate G_i of C in topological order, label G_i with its index $label(G_i) = i$, and then distinguish the cases:

   - If G_i is an *XOR-gate* $W_c = XOR(W_a, W_b)$ with garbled input values $w_a^0 = \langle k_a^0, p_a^0 \rangle, w_b^0 = \langle k_b^0, p_b^0 \rangle, w_a^1 = \langle k_a^1, p_a^1 \rangle, w_b^1 = \langle k_b^1, p_b^1 \rangle$, then set the garbled output values as follows:

$$w_c^0 = \langle k_a^0 \oplus k_b^0, p_a \oplus p_b \rangle \quad (3.3)$$
$$w_c^1 = \langle k_a^0 \oplus k_b^0 \oplus R, p_a \oplus p_b \oplus 1 \rangle \quad (3.4)$$

- If G_i is a 2-input gate $W_c = g_i(W_a, W_b)$ with garbled input values $w_a^0 = \langle k_a^0, p_a^0\rangle, w_b^0 = \langle k_b^0, p_b^0\rangle, w_a^1 = \langle k_a^1, p_a^1\rangle, w_b^1 = \langle k_b^1, p_b^1\rangle$, then:
  i. Choose a random garbled output value corresponding to zero as $w_c^0 = \langle k_c^0, p_c^0\rangle \in \{0,1\}^{N+1}$.
  ii. Set the garbled output value corresponding to one as $w_c^1 = \langle k_c^1, p_c^1\rangle = \langle k_c^0 \oplus R, p_c^0 \oplus R\rangle$.
  iii. Create the garbled table corresponding to these gate values. For each of the 2^2 possible combinations of G_i's input values $v_a, v_b \in \{0,1\}$ set

$$e_{v_a,v_b} = H(k_a^{v_a}||k_b^{v_b}||i) \oplus w_c^{g_i(v_a,v_b)}, \tag{3.5}$$

  where $H : \{0,1\}^* \mapsto \{0,1\}^{N+1}$ is a random oracle. Then sort the entries e in the table by the input pointers.
- For each output wire W_i with garblings $w_i^0 = \langle k_i^0, p_i^0\rangle$ and $w_i^1 = \langle k_i^1, p_i^1\rangle$, create a garbled table for both possible wire values $v \in \{0,1\}$. Set the entries $e_v = H(k_i^u||\text{“}out\text{”}||j) \oplus v$ and sort them by the input pointers. i.e. place entry e_v in position p_i^v.

All garbled tables and garblings of the constructor's input values are transferred to the evaluator by OT. The corresponding garbled circuit evaluation algorithm, run by the evaluator-participant, is the following:

1. For each input wire W_i of the circuit, receive the corresponding garbled value $w_i = \langle k_i, p_i\rangle$
2. For each gate G_i
   - If G_i is an XOR-gate with garbled input values $w_a = \langle k_a, p_a\rangle$, $w_b = \langle k_b, p_b\rangle$, then compute the garbled output value $w_c = \langle k_c, p_c\rangle = \langle k_a \oplus k_b, p_a \oplus p_b\rangle$.
   - If G_i is a 2-input gate other than XOR with garbled input values $w_a = \langle k_a, p_a\rangle$, $w_b = \langle k_b, p_b\rangle$, then decrypt the garbled output value fromt he garbled table entry e in position $\langle p_a, p_b\rangle : w_c = \langle k_c, p_c\rangle = H(k_a||k_b||i) \oplus e$.
3. For each output wire W_i with garbling $w_i = \langle k_i, p_i\rangle$ decrypt the output value f_i from the garbled output table entry e in row p_i as follows:

$$f_i = H(k_i||\text{“}out\text{”}||j) \oplus e \tag{3.6}$$

   where $f : \{0,1\}^{v_1} \times \{0,1\}^{v_2} \to \{0,1\}^v$.

In order to make clear how this concept works, the authors in [KS08] described an improved SFE implementation of the XOR gate as follows:

Example 3.1.2 *Let P_1 be the constructor of the circuit with private input $x = \langle x_1, x_2, ...x_{v_1}\rangle$ and P_2 be the evaluator of the circuit, with private input $y = \langle y_1, y_2, ...y_{v_2}\rangle$. Let G be a gate of the circuit that has two input wires W_a, W_b and one output wire W_c.*
In order to garble the wire values, at first P_1 should randomly choose $w_a^0, w_b^0, R \in_R \{0,1\}^N$. Then the garbled values for the output wire are determined as follows :

$$w_c^0 = w_a^0 \oplus w_b^0 \quad w_i^1 = w_i^0 \oplus R \quad \forall i \in \{a, b, c\}. \tag{3.7}$$

The restriction that this construction imposes on the garbled values, is that the garblings w_i^j of the two values of the same wire are not chosen independently, but must differ by the same global value R. The garbled gate output is obtained by XORing the garbled gate inputs:

$$\begin{aligned} w_c^0 &= w_a^0 \oplus w_b^0 = (w_a^0 \oplus R) \oplus (w_b^0 \oplus R) \\ w_c^1 &= w_c^0 \oplus R = w_a^0 \oplus (w_b^0 \oplus R) = w_a^0 \oplus w_b^1 = (w_a^0 \oplus R) \oplus w_b^0 = w_a^1 \oplus w_b^0 \end{aligned}$$

The constructor P_1 calculates these values for each gate, according to the type of the gate. Then he creates the garbled tables following the steps in the first algorithm and he sends them together with the garbled values w_i^j corresponding to his private input $\{0,1\}^{v_1}$, through OT to the evaluator P_2. Then P_2 uses his input values and the garbled tables to retrieve the output value f_i. According to the type of the gate that is used every time (P_2 knows the topology of the circuit) and the garbled tables, he computes or decrypts the garbled output value and finally he can obtain the function $f(x, y)$, which relates his inputs with the private input values of the constructor.

Although it is a quite simple procedure to obtain the output values from the above mentioned formulas, it is not possible to reveal the wire values from the corresponding garblings. The outputs of a random oracle H are used as one-time pad to encrypt the garbled output values in the garbled tables, ensures the randomness of the results. Any specific combination of H's inputs is used for encrypting at most one table entry throughout the whole circuit construction. Moreover, since the second player who is going to decrypt one message, knows only one value per garbled wire, he can decrypt exactly one entry of G_i's garbled table. All other entries are encrypted with keys that cannot be guessed by an evaluator in polynomial time. In this way, one of the two output values for each wire looks random to him. Therefore, the security of the circuit is preserved. A complete proof of security for this protocol is given in [Kol05].

3.2 Arithmetic Circuit Approach - Secret Sharing

The arithmetic circuit approach to secure multiparty computations is based on secret sharing and operates on an arithmetic circuit representation of the computed

function. Protocols following this approach can be efficiently applied when there exists an honest majority among the participants and when the reconstruction of the secret involves addition or multiplication operations on the shares. VIFF is a useful platform for developing such protocols and it has been used already to develop real-life applications of secure multiparty computations, such as the SIMAP project as reported in [BCD+08], Distributed RSA, Distributed AES, and secure voting systems.

Secret sharing is a method for distributing a secret amongst n participants. Each participant holds a *share* of the secret and it can only be reconstructed when a sufficient number of them cooperates. Secret sharing was invented by Adi Shamir and George Blakley independently in 1979. In [Sha79] the problem of secret sharing is defined as the problem in which the secret is some data D (e.g., the safe combination) and in which nonmechanical solutions are also allowed. The goal is to divide D into l pieces $D_1, ..., D_l$ in such a way that:

1. Knowledge of any t or more D_i pieces makes D easily computable;
2. Knowledge of any $t - 1$ or fewer D_i pieces leaves D completely undetermined (in the sense that all its possible values are equally likely).

Such a scheme is called a (t, l)**-threshold scheme**. The critical step in "splitting a secret" is to use some additional randomness, so that individual sharings or combination of less than $t-$participants reveal nothing about the secret.

In [Sch05], a secret sharing scheme is defined as an interaction between a dealer D and participants $P_1, ..., P_l$ by using the following protocols:

- **Distribution:** A protocol in which the dealer D shares a secret s such that each participant P_i obtains a share $s_i \quad \forall 1 \leq i < l$.
- **Reconstruction:** A protocol in which the secret s is reconstructed by pooling shares $s_i, P_i \in A$ for any qualified set of participants $A \subseteq \{P_1, ..., P_l\}$.

Access Structure is the set Γ of all qualified subsets of $\{P_1, ..., P_l\}$. A requirement is that an access structure Γ is monotone, in the sense that any qualified set remains qualified when participants are added to it. If $A \in \Gamma$ is the qualified set, then $A \cup \{P_i\} \in \Gamma$ as well. The access structure of a $(t, l)-$threshold scheme is defined as $\Gamma = \{A \subseteq \{P_1, ..., P_l\} : |A| \geq t\}$ for $1 \leq t \leq l$. Two other requirements are imposed on secret sharing schemes. These are *correctness*, in the sense that any qualified set of participants can determine the secret value s by combining their shares, and *privacy*, so that any non-qualified set of participants cannot determine any information about s.

3.2.1 Shamir Secret Sharing

Shamir proposed a $(t, l)-$threshold secret sharing scheme, $1 \leq t \leq l$, which can be applied whenever the secret belongs to a finite field $\mathbb{F}_q$ of order q with $q > l$. Shamir's scheme will be described for the case $q = p$, where p is a prime, because in this way the elements of $\mathbb{F}_q = \mathbb{Z}_p$ can be treated as integers modulo p. Shamir's $(t, l)-$threshold secret sharing scheme can be described as follows:

- **Distribution:** The dealer picks a random polynomial $r(x)$ of degree $< t$ satisfying $r(0) = s$. That is,

$$r(x) = s + r_1 x + r_2 x^2 + ... + r_{t-1} x^{t-1} \tag{3.8}$$

  where $r_1, ..., r_{t-1} \in_R \mathbb{Z}_p$. For $i = 1, ..., l$ share $s_i = r(i)$ is sent to participant P_i through a private channel.

- **Reconstruction:** Any qualified set A of t participants can recover the secret s from their shares by Lagrange interpolation:

$$s = \sum_{i \in A} s_i \lambda_{A,i} \quad with \quad \lambda_{A,i} = \prod_{j \in A \setminus \{i\}} \frac{j}{j-i} \tag{3.9}$$

Correctness and privacy of this scheme is proven in [Sch05]. We hereby give an example of how the protocol works, in order to familiarize the reader with the concept of secret sharing that is important for understanding the next chapter.

Example 3.2.1 *Suppose that we have a $(3, 4)-$ threshold scheme with 4 participants P_1, P_2, P_3, P_4, who want to share the secret $s = 3$. Let the dealer pick the polynomial $r(x) = 2x^2 + 4x + 3$ with elements in $\mathbb{Z}_5$. At first, we note that this is a proper Shamir scheme, because all the appropriate properties hold:*

- $t = 3, l = 4$ & $1 \leq t \leq l \Rightarrow 1 \leq 3 \leq 4$ & $deg(r(x)) = 2 < 3$
- $q > l \Rightarrow 5 > 4$
- $r(0) = s$ *and indeed* $r(0) = 3$

The dealer sends the following to each participant:

- *To P_1 he sends $s_1 = r(1) = 2 + 3 + 4 = 9 \Rightarrow s_1 = 4$ mod 5*
- *To P_2 he sends $s_2 = r(2) = 8 + 8 + 3 = 19 \Rightarrow s_2 = 4$ mod 5*
- *To P_3 he sends $s_3 = r(3) = 18 + 12 + 3 = 33 \Rightarrow s_3 = 3$ mod 5*
- *To P_4 he sends $s_4 = r(4) = 32 + 16 + 3 = 51 \Rightarrow s_4 = 1$ mod 5*

Since this is a $(3,4)-$ threshold scheme, the combination of the shares of 3 participants are enough to recover the secret. So the access structure A has cardinality $|A| = 3$, and we assume that $A = \{P_1, P_2, P_3\}$. From equation 3.9, we have the following:

$$\lambda_{A,1} = \prod_{j \in A \setminus \{1\}} \frac{j}{j-1} = \frac{2}{2-1}\frac{3}{3-1} = 2 \cdot \frac{3}{2} = 3 \quad (3.10)$$

$$\lambda_{A,2} = \prod_{j \in A \setminus \{2\}} \frac{j}{j-2} = \frac{1}{1-2}\frac{3}{3-2} = (-1) \cdot 3 = -3 \quad (3.11)$$

$$\lambda_{A,3} = \prod_{j \in A \setminus \{3\}} \frac{j}{j-3} = \frac{1}{1-3}\frac{2}{2-3} = \frac{-1}{2} \cdot (-2) = 1 \quad (3.12)$$

Following the reconstruction equation 3.9, the secret $s = 3$ can indeed be retrieved:

$$s = \sum_{i \in A} s_i \lambda_{A,i} = s_1 \lambda_{A,1} + s_2 \lambda_{A,2} + s_3 \lambda_{A,3} = 4 \cdot 3 + 4 \cdot (-3) + 3 \cdot 1 = 3$$

3.2.2 Verifiable Secret Sharing

The basic secret sharing scheme that was described in the previous section is defined to resist against passive attacks, since it is based on the assumption that all participants run the protocol according to the scheme. However, in many applications, including secure multiparty computations that we are interested in, a secret sharing scheme is also required to withstand active attacks. Active attacks can include a cheating dealer, who sends incorrect shares to the participants during the distribution protocol, or cheating participants, who during the reconstruction protocol send different shares from the ones that they obtained [Sch05]. A verifiable secret sharing protocol that can withstand such attacks can be defined as follows:

Definition 3.2.1 *A* ***verifiable secret sharing*** *(VSS) scheme includes auxiliary information for each participant, so that they can verify their shares as consistent, even if the dealer is malicious, while in a secret sharing scheme such information are omitted, because the dealer is supposed to be honest.*

In [Gol04] VSS is defined in terms of secure multiparty computation as a secure multi-party protocol for computing the randomized functionality corresponding to some (non-verifiable) secret sharing scheme. This definition is stronger than other definitions and is very convenient to use in the context of general secure multi-party computations.

The auxiliary information needed at each phase of the VSS protocol are the following:

- **Distribution:** During this protocol, dealer D shares a secret s, such that each participant P_i obtains a share s_i. Each participant is able to decide whether his share is correct or not, according to some additional information, which show that the dealer is indeed committed to the value that he claims to share.
- **Reconstruction:** At this phase, the secret s is reconstructed by combining the shares s_i of each participant belonging in any qualified set $A \subseteq \{P_1, ..., P_l\}$. Each share is accompanied by a proof that shows whether the share is correct or not.

In [GRR98] an efficient VSS protocol was proposed, which satisfies the VSS definition 3.2.1 and it is based on Shamir secret sharing with an additional low cost added construction that is necessary for the security of the scheme. This protocol works as follows: The dealer chooses two random polynomials $f(x)$ and $r(x)$ of degree t. The constant term of $f(x)$ will be secret shared. The polynomial $r(x)$ will be used to generate $t-$wise independent random strings that will be used to commit to the secret. Each player P_i receives $f(i)$, which is his share to the secret, and $r(i)$ which is the randomness associated with him. The dealer should commit to the shares of players by broadcasting $C(f(i), r(i))$, where C is a commitment function. This commitment function can be a hash function, for instance SHA-1(f(i),r(i)). In [GRR98] the correctness and security of this protocol were proven.

Addition of secrets f_0 and g_0 corresponding to the constant terms of the polynomials $f(x)$ and $g(x)$ respectively, can be performed locally in a similar way to the example described previously. Each player P_i can follow that procedure twice and then add his shares to evaluate $f(i) + g(i)$. However, multiplication of two secrets, which are distributed among the players cannot be performed locally, since the polynomial generated after multiplication will be of higher degree. Let f_0 and g_0 be two secrets shared by polynomials $f(x)$ and $g(x)$ of degree t. The players would like to compute the product $f_0 \cdot g_0$. The constant term of the polynomial $f(x) \cdot g(x)$ will the the secret that they want to share, but the degree of $f(x) \cdot g(x)$ will be $2t$. Therefore, it is not sufficient for each player to locally multiply his shares of both secrets. To overcome this issue, reduction and randomization protocols are needed. In [BOGW88] such protocols were proposed and later these two protocols were implemented in one step by [Rab81]. We are going to describe briefly this proposal for the case where all the players act properly. In this paper the security of this protocol is proven against a polynomial time adversary under the Discrete Logarithm assumption.

Let $f(i)$, $g(i)$ be the shares of player P_i on $f(x)$ and $g(x)$ respectively. The product of $f(x)$ and $g(x)$ is:

$$f(x)g(x) = c_{2t}x^{2t} + ... + c_1 x + c_0 =^{def} fg(x) \tag{3.13}$$

where $c_0 = f_0 \cdot g_0$. We define $fg(i) = f(i)g(i) \forall 1 \leq i \leq 2t+1$ and thus we have the following:

$$A \begin{vmatrix} c_0 \\ c_1 \\ . \\ . \\ . \\ c_{2t} \end{vmatrix} = \begin{vmatrix} fg(1) \\ fg(2) \\ . \\ . \\ . \\ fg(2t+1) \end{vmatrix}$$

$A = (a_{ij})$ is a $(2t+1)(2t+1)$ Van der Monde matrix defined by $a_{ij} = i^{j-1}$ and can be easily inverted. Let the first row of the inverse matrix A^{-1} be $(\lambda_1, ..., \lambda_{2t+1})$. These constant values are knows to all the participants. Then, equation 3.13 implies that $\lambda_1 fg(1) + ... + \lambda_{2t+1} fg(2t+1)$. Define $H(x) =^{def} \sum_{i=1}^{2t+1} \lambda_i h_i(x)$, where $h_i(x)$ are polynomials of degree t which satisfy $h_i(0) = fg(i) \quad \forall 1 \leq i \leq 2t+1$. It holds:

$$H(0) = \lambda_1 fg(1) + ... + \lambda_{2t+1} fg(2t+1)$$

$$H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j) \tag{3.14}$$

and $H(0)$ is exactly the product $c_0 = f_0 g_0$. Thus, if every player P_i shares his part of the secret using a polynomial $h_i(x)$ with the properties as defined above, then the polynomial $H(x)$ can be used to share the secret and it is of degree t. Thus, the degree reduction is achieved. Moreover, $H(x)$ is random, since the coefficients λ_i are non-zero and it consists of $n-t$ polynomials $h_i(x)$, which are random if they are chosen by good players. in this way, the secret $f_0 g_0$ can be shared by using a random polynomial of degree t.

3.2.3 Pseudorandom Secret Sharing

The generation of random shared values is useful in secure multi-party protocols, and sometimes it needs to be done locally, in order to minimize the communication cost. *Pseudorandom Secret Sharing*(PRSS) offers the advantage of calculating the shared values locally, at the cost of losing perfect security. In a PRSS protocol every minimal qualified set of players knows a secret key or a pseudorandom function used to compute a random value on some public input. At the end, the random shared value will be the sum of all those values.

A first approach to local conversion of secretely distributed values r_{A_i} to shares of a secret s is the following:

- Let $s = \sum_{A \subseteq [n]:|A|=n-t, 1 \leq i \leq n-t} r_{A_i}$ be the secret that we want to share, where r_{A_i} is given to all players in the set A.

- For every set $A \subseteq [n], with |A| \leq n-t$, let f_A be the unique degree-t polynomial, such that
  1. $f_A(0) = 1$ and
  2. $f_A(i) = 0 \quad \forall i \in [n] \backslash A$.
- Each player P_i can locally compute its share s_i as follows:

$$s_i = \sum_{A \subseteq [n], |A|=n-t, i \in A} r_{A_i} \cdot f_A(i) \tag{3.15}$$

It can be verified that the shares s_i correspond to the share $f(0) = s$: Define a polynomial $f = \sum_{A \subseteq [n]:|A|=n-t, i \in A} r_{A_i} \cdot f_A$. Then f has degree at most t, condition (1) on the $f_A' s$ implies that $f(0) = s$ and condition (2) implies that $f(i) = s_i$.

For the pseudorandom setting, we first note that the initially distributed values r_{A_i} can be used as keys to a pseudorandom function $\psi(\cdot)$, since the shares r_{A_i} will be random and independent, when the secret s is random. A PRSS scheme can be analyzed in the following steps:

1. The players should at first agree on a common input value α to the function ψ, they can all compute $\psi_{r_{A_i}}(\alpha)$ and use these values in the secret sharing scheme to replace the role of r_{A_i}.
2. Now each player can compute its share s_i as

$$s_i = \sum_{A \subseteq [n]:|A|=n-t, i \in A} \psi_{r_{A_i}}(\alpha) \cdot f_A(i) \tag{3.16}$$

which indeed equals the values of the shares in equation 3.15, whereby r_{A_i} have been replaced by $\psi_{r_{A_i}}(\alpha)$.

Chapter 4

Secure Multiparty AES

4.1 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a symmetric-key encryption standard that was announced as Federal Information Processing Standards Publications (FIPS PUBS) by the National Institute of Standards and Technology (NIST) in 2006 and was adopted by the US Government to protect sensitive information. Two Belgian cryptographers, Vincent Rijmen and Joan Daemen developed the Rijndael algorithm, which selected to become the AES by NIST in 2000.

In the following sections, a brief description of the algorithmic specifications and the implementation is given. More detailed information about the standard can be found in the original document issued by NIST [Pub01].

4.1.1 Algorithm Specifications

The AES algorithm accepts as input a fixed block size of 128 bits, encrypts them by using a cipher Key of 128, 192 or 256 bits, and then produces a block of 128 bits as output. The operations are performed on a 4×4 array of bytes, called the *State*. The state array is used as an intermediate step to facilitate the transition from the initial input array to the output array. It is used for both the encryption and the decryption phase. The encryption algorithm is called **Cipher** and consists of all the transformations that convert the plaintext to ciphertext using the **Cipher Key**. The inverse procedure for the decryption is called **Inverse Cipher**.

At the start of the cipher, the input is copied to the state array according to the scheme:

$$s[r,c] = in[r+4c] \, \forall \, 0 \leq r < N_b \quad and \quad 0 \leq c < N_b \tag{4.1}$$

where s is the state array, in is the input array, r indicates the row number and c the column number. N_b is the number of columns ($32 - bit$ words) of the state, which equals 4 for this standard. After applying some transformations, the ciphertext appears in the output array out according to the scheme:

$$out[r + 4c] = s[r, c] \, \forall \, 0 \leq r < N_b \quad and \quad 0 \leq c < N_b \tag{4.2}$$

The main function of the AES algorithm is a round function consisting of the following byte-oriented transformations:

1. `SubBytes()` Transformation
   The SubBytes transformation is performed on each byte of the state array using a substitution table called **S-Box**. The S-Box is an invertible table that is constructed by taking the multiplicative inverse of every element in the field $GF(2^8)$ (the zero element is mapped to zero) and then applying the following affine transformation over $GF(2)$:

   $$b_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i \, \forall \, i \in \{0, ..., 7\} \tag{4.3}$$

   where b_i is the ith bit of the byte and c_i is the ith bit of the constant vector value $c = \{11000110\}$.

   The following figure depicts the above mentioned procedure, whereby each element a_{ij} of the first matrix represents one byte and is transformed to the corresponding element b_{ij} after an inversion of the initial element and an affine transformation using the constant vector value c.

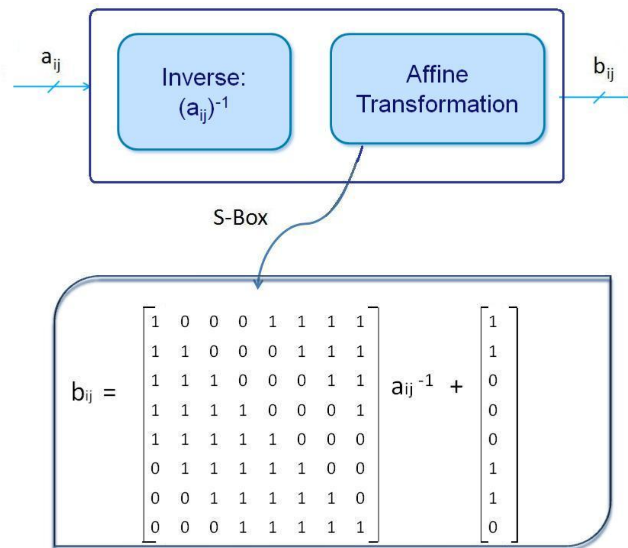

Figure 4.1: SubBytes transformation

2. `ShiftRows()` Transformation
   This transformation permutes cyclically the bytes in the last three rows of the state array according to the formula:

$$s_{r,c} = s_{r,(c+shift(r,N_b) mod N_b)} \quad \forall 0 < r < 4 \quad and \quad 0 \leq c < N_b \tag{4.4}$$

   The value $shift(r, N_b)$ depends on the row number and it is:

$$shift(1,4) = 1 \quad shift(2,4) = 2 \quad shift(3,4) = 3 \tag{4.5}$$

   The first row $r = 0$ is not shifted.

3. `MixColumns()` Transformation
   In the MixColumns Transformation each column of the state array is treated as a four term polynomial over $GF(2^8)$ and it is multiplied with the fixed polynomial modulo $x^4 + 1$:

$$a(x) = 03x^3 + 01x^2 + 01x + 02 \tag{4.6}$$

4. `AddRoundKey()` Transformation
   During this transformation, a Round Key is added to the state by a bitwise XOR operation. Each Round Key is produced by the Key Expansion routine and depends on N_b words.

The number of rounds N_r that are performed during the execution of the AES algorithm depends on the key size. For a Cipher Key of length 128 bits $N_r = 10$, for a key of length 192 $N_r = 12$ and for a key of length 256 the number of rounds will be 14. After the appropriate number of rounds is completed, the final state array is copied to the output array according to the scheme 4.2.

The **Key Expansion** routine, which is used for the AddRound Key transformation, performs an expansion of the Cipher Key C, in order to generate different keys for each round. The AES algorithm requires an initial set of N_b words and each of the N_r rounds requires N_b words of key data, so in total $N_b + N_b \cdot N_r = N_b(N_r + 1)$ words are required. Therefore, the key expansion routine generates w_i keys, with $i \in \{0, ...N_b(N_r + 1) - 1\}$. More precisely,

$$w_0 = CipherKey, \qquad w_i = w_{i-1} \oplus w_{i-N_k} \tag{4.7}$$

where w_{i-N_k} is the sequence of bytes N_k positions earlier than i.

For decrypting a message with AES, the Inverse Cipher algorithm is used, which consists of exactly the same transformations as in the Cipher algorithm implemented in reverse order.

4.1.2 Implementation Issues

Implementation of the AES algorithm can vary according to the platform that is used and the performance that we need to achieve. All the implementations that produce the same ciphertext, given the same plaintext and cipher key, are accepted by the standard that specifies AES. However, there are some specified allowed values that the standard explicitly defines for AES. The key length should be 128, 192 or 256 bits long and according to that the block size and the number of rounds should be fixed. Apart from these parameterized values, there are no other restrictions on the key selection specified by the standard, since no weak keys have been identified so far.

One of the most important implementation issues of AES algorithm, which is also the main part of this research, is the design of the S-box, in order to achieve high performance. The inversion of the elements over $GF(2^8)$ is the most expensive operation that should take place in the S-Box. Every field element represented by the byte is inverted in $GF(2^8)$, except from 0 that is mapped to 0. The inversion algorithm used in each method involves multiplications over $GF(2^8)$, which require interaction between the participants in the multiparty case. The multiparty AES implementation will be described in detail later on in this chapter. Many methods have been proposed in the literature for efficient inversion algorithms and they are going to be presented in the following sections. These methods are going to be analyzed from another point of view, namely the number of $GF(2)$ operations needed for the inversion process, which will give us the opportunity to investigate the efficiency and complexity of these methods in a lower, bit-wise level.

The other three transformations taking place in the encryption or decryption algorithms, apart from the substitution tables, are fixed procedures that consist of permutation of bits and XOR operations. In what follows, improvements on these transformations are not going to be further investigated, since they do not require the use of expensive operations.

4.2 Multiparty Computation Protocols for AES

Secure Multiparty computations in terms of cryptographic algorithms are based on the fact that the secret key, the plaintext and the outputted ciphertext are secret shared among the players. For AES algorithm these values are byte-wise secret shared over $GF(2^8)$. An interesting application of such a threshold approach to symmetric encryption would be in environments, where data are stored in places with weak security compliances like in cloud computing. The participants would run a MPC protocol to securely compute a secret shared AES encrypted ciphertext

of a secret shared plaintext with a secret shared AES key K.

The general categorization of MPC protocols in the binary and arithmetic circuit approach analyzed in *Chapter 3* appears also in MPC protocols for AES. In the following sections, the secure two-party AES implementation of [PSSW09] will be presented, which is based on Yao's garbled circuits. The most efficient approach was introduced by Satoh et al in [SMTM01], whereby $8-$bit calculations are reduced to $4-$bit ones and then further to $2-$bit using the "tower field" approach. In this way, at the lowest field we can apply Yao's technique and by the free-XOR gate construction, the XOR gates can be evaluated for free. On the other hand, the implementations of secure multiparty AES are based on the arithmetic properties of AES in $GF(2^8)$. Secure multiplication and addition in $GF(2^8)$ can be implemented by Shamir secret sharing with polynomials chosen on appropriate defined basis. The main issue of our research is the efficient implementation of the AES S-box, which requires the secure computation of the multiplicative inverse of the input elements in $GF(2^8)$.

According to Boyar and Peralta [BP10], the new term of multiplicative complexity can be defined as follows:

Definition 4.2.1 *The **multiplicative complexity** of a function is the number of $GF(2)$ multiplications necessary and sufficient to compute it.*

In this setting, we are interested in reducing the multiplicative complexity of the AES algorithm and more precisely, the non-linear components of the AES S-Box. Boyar and Peralta introduced the smallest and most compact AES circuit up-to-date following Satoh's and Canright's approach, as we are going to see in the next section.

4.2.1 Secure Multiparty AES

In this section, an efficient algorithm for secure multiparty AES in the arbitrary case is going to be presented. The key, the plaintext and the ciphertext are supposed to be bytewise secret shared among the players. This approach is usually applied when there is an honest majority among the participants and it handles addition and multiplication operations more efficient than the binary circuit approach.

4.2.1.1 Masked Exponentiation

In [DK10] three methods are proposed for the inversion of a field element in $GF(2^8)$ in the multiparty computation setting of implementing the S-Box of AES. The most

efficient method to calculate the multiplicative inverse x^{254} of an element x over $GF(2^8)$ is *"Masked Exponentiation"*. The main idea of this method is to mask the secret shared value $[x]$ by a random shared value $[r]$, both in $GF(2^8)$, open it and exponentiate locally. The binary powers of both values, the opening $(x + r)$ and the random $[r]$, are calculated by using the fact that squaring is a linear operation in fields with characteristic 2, that is $(x + y)^2 = x^2 + 2xy + y^2 = x^2 + y^2$. More generally, this property can be expressed as follows:

$$(x+y)^{2^i} = ((x+y)^{2^{i-1}})^2 = (x^{2^{i-1}} + y^{2^{i-1}})^2 = x^{2^i} + y^{2^i} \quad \forall i \geq 2$$

The binary powers of the random shared value $[r] \in GF(2^8)$ can be calculated in a preprocessing phase in parallel for all the S-boxes of the protocol, and the powers of the openings can be computed locally.

$$\begin{aligned} [r^2] &= [r] \cdot [r], [r^4] = [r^2] \cdot [r^2], ..., [r^{128}] = [r^{64}] \cdot [r^{64}] \\ open([x] + [r]) &= (x+r), \quad (x+r)^2, (x+r)^4, ..., (x+r)^{128} \end{aligned}$$

The binary powers of $[x^{2i}]$ are the sum of the previous values, which are then multiplied securely to obtain $[x^{254}]$. The secure multiplication can be performed as a verifiable secret sharing scheme, as explained in *Chapter 3*.

$$(x+r)^{2^i} + [r^{2^i}] = [x^{2^i}] \quad \forall i = 1, ..., 7 \tag{4.8}$$

$$\prod_{i=7}^{7} [x^{2i}] = [x^{\sum_{i=1}^{7} 2^i}] = [x^{254}] \tag{4.9}$$

Complexity

The communication complexity of these methods and the complete implementation of this algorithm is quite small compared to the methods of the binary approach, due to the utilization of the arithmetic properties of AES and the non dependance on the number of participants. The online operations are 1 opening and 6 secure multiplications in 4 rounds. The secure multiplications can be performed by using the method described in *Section 3.2.2*. Furthermore, for the calculation of the binary powers of $[r]$ 7 multiplications are needed in 7 rounds, but they can be performed in a preprocessing phase. In the following section, we present an efficient way to calculate $[r^{128}]$ with only 63 additions, without the need to perform any secure multiplication. Since additions need no interaction, these calculations can be performed online in an efficient way.

4.2.2 Binary Circuit Approach - Secure Two-Party AES

Pinkas et al. described an implementation of the two-party AES algorithm using garbled circuits and presented some algorithmic protocol improvements, which made

it feasible to implement large circuits, such as AES, in practice [PSSW09]. In their AES circuit, they assumed that the input of party P_1 is the 128−bit secret key, the input of party P_2 is the 128−message block and the requirements of the system is that P_2 learns the encryption of its message under P_1's secret key, while P_1 learns nothing.

The secure function evaluation for AES encryption made use of Yao's garbled circuit construction, described in the previous chapter. The resulting circuit computed an AES encryption of a single block using the general optimization methods presented in [PSSW09] and it was compiled using the Fairplay compiler. The encryption scheme of this approach is implemented via:

$$E^s_{k_1,k_2}(m) = m \oplus KDF^{|m|}(k_1, k_2, s) \tag{4.10}$$

where KDF is a key derivation function, whose $|m|$ bits of output are independent of the two input keys and depends on the value of s. The key derivation function is instantiated as follows:

$$KDF^l(k_1, k_2, s) = H(k_1||s)_{1...l} \oplus H(k_2||s)_{1...l} \tag{4.11}$$

In this paper, the hash function H was implemented by using SHA-256. The assumption that the function H behaves as a pseudo-random function is not enough for the proposed optimizations; the stronger assumption that the function is correlation robust is needed.

Definition 4.2.2 *An efficiently computable function $H : \{0,1\}^* \to \{0,1\}^l$ is correlation robust if the following distribution is pseudorandom:*

$$(t_1, ..., t_m, H(t_1 \oplus r), ..., H(t_m \oplus r)),$$

where $t_1, ..., t_m$ and r are chosen at random, and m is polynomial in the security parameter.

This assumption has been introduced in [CH08] and was used to provide security against malicious adversaries for a method of oblivious transfer. To clarify this assumption, it is mentioned that highly efficient protocols can be constructed, if we assume that all the pads that are used for encrypting the table entries are pseudorandom.

The free-XOR trick of Kolesnikov & Schneider [KS08] was used here, in which the XOR gates are evaluated for free and there is no need to evaluate or transmit the garbled tables for such gates. An optimization of this technique requires that there is a global random value R of bit length t, known only to P_1, such that for all

garbled wires w_i it holds that $k_i^1 = k_i^0 \oplus R$. So the garbling of the 1 value of a wire is determined from XOR-ing the garbled 0 value with the value R. In order to reduce the size of the tables of the non-XOR gates, the *Garbled Row Reduction (GRR)* method was proposed in [PSSW09]. For the AES circuit, 2-to-1 gates were used and therefore, we assume that the circuit consists of gates with two input wires w_1, w_2 and one output wire w_3. The input to w_1 is denoted by b_1 and is known to P_1, while the input to w_2 is b_2 and it is known to P_2. Each gate has a unique identifier G_{id}, which enables a circuit fan out of greater than one, i.e., it enables the output wire of one gate to be used in more than one other gate. We require that P_2 evaluates the gate on the two inputs, without P_1 learning anything, and without P_2 determining the value b_1, i.e. P_2 learns the ciphertext under P_1's secret key and P_1 learns nothing. We define the output of the gate by the function $G(b_1, b_2) \in \{0, 1\}$. The construction of the circuit according to Yao proceeds as usual, with P_1 the constructor and P_2 the evaluator of the circuit. The GRR method refers to the garbled values of the output wires. Instead of defining randomly the two garbled values of the output wires, they proposed the definition of one garbled value as a function of garbled values of the two input wires, which will result in this output value. More precisely, an input pair $(b_1, b_2) \in \{0, 1\}^2$ is chosen and by applying the function $G(b_1, b_2)$ to this pair, we get the garbled values of b_1, b_2. In this way, these values need not be stored in the gate table. In the evaluation phase, if the evaluator has the garbled values of the pair (b_1, b_2) it can compute the corresponding garbled output directly, without consulting the gate table.

Let k_i^0, k_i^1 denote the garbles wire values as in Yao's approach, $G(b_1, b_2)$ the function being implemented by the gate and set the external value of the wire to be $c_i = \pi_i(b_i)$, where π_i is a random permutation of $\{0, 1\}$ associated to each wire. Then, the garbled output value corresponding to the output that comes from the external input values $(c_0, c_1) = (0, 0)$ is defined as:

$$k_3^{G(\pi_1^{-1}(0),\pi_2^{-1}(0))}||c_3 = KDF^{t+1}(k_1^{\pi_1^{-1}(0)}, k_2^{\pi_1^{-1}(0)}, G_{id}||0||0). \tag{4.12}$$

The garbled value is exactly equal to the pseudo-ranodm mask that was used to hide it in the basic protocol. This operation also defines the external value c_3 of this output value. The permutation π_3 can be defined in a way, such that $c_3 = \pi_3(G(\pi_1^{-1}(0), \pi_2^{-1}(0)))$. The other garbled value of the output wire is chosen as in the free-XOR method:

$$k_3^{G(\pi_1^{-1}(0),\pi_2^{-1}(0))} = k_1^{G(\pi_1^{-1}(0),\pi_2^{-1}(0))} \oplus k_2^{G(\pi_1^{-1}(0),\pi_2^{-1}(0))} \tag{4.13}$$

In this way, the first entry of the garbled table can be evaluated by the participant, without the need to store or transmit it, and therefore, a 25% reduction in the number of non-XOR gates is achieved. The evaluator of the garbled gate proceeds as in the standard algorithm for the rest of the garbled inputs. Only when the external values of both input wires are 0, $c_1 = c_2 = 0$, then he uses the garbled values

$k_1^{b_1=\pi_1^{-1}(0)}$ and $k_2^{b_1=\pi_2^{-1}(0)}$ to compute $k_3||c_3$ as described previously. This observation concludes the GRR optimization.

More details on the implementation and other optimizations of this circuit can be found in [PSSW09]. The main issue in this approach is that by assuming that the KDF is correlation robust then the GRR optimization produces the most efficient implementation for such large circuits like AES.

4.2.2.1 Complexity

Pinkas et al. derived a circuit of 33880 2-to-1 gates for the AES algorithm by constructing it according to Yao's garbled circuits approach and applying some optimizations, such as the GRR technique. It is mentioned that 66% of these gates are XOR gates, whose results can be computed using the free-XOR trick of Kolesnikov. Therefore, 22361 gates do not affect the performance of the computations. Moreover, 25% of the non-XOR gates can be evaluated for free by using the GRR method, i.e. $(33880 - 22361) * 25\% = 2880$ gates are also saved. This results in the remaining 8639 gates that affect the computational workload of the operations.

4.2.3 Straightforward methods of Constructing the AES S-Box

Constructing a compact and efficient binary circuit to calculate the result of the S-Box in the AES algorithm is not easy to achieve, because it is a non-linear transformation. An S-Box is the multiplicative inverse of every byte-wise element of the state array over the field $GF(2^8)$ followed by an affine transformation. Each of the 16 elements of the state array are bytes, so they can be represented as polynomials in $GF(2^8)$ with coefficients in $GF(2)$, equal to the 8 bits of each byte. For instance, the element $a = \{11011001\}$ can be written in the polynomial format as follows: $a = x^7 + x^6 + x^4 + x^3 + 1$, where the least significant bit will be the constant term of the polynomial in $GF(2^8)$. As it is mentioned in Chapter 2, the multiplicative inverse of every element α in this field is α^{254} reduced by the irreducible polynomial that is chosen for this field.

In this section, we are going to present two straightforward implementations of the AES S-Box, in order to show the degree in which the inversion over $GF(2^8)$, the only non-linear component of the S-Box, affects the computational workload of this operation. At first, we are going to discuss a standard implementation of the AES S-Box using the field arithmetic in $GF(2^8)$ and then some improvements that can be achieved by the Karatsuba algorithm, which can reduce the multiplicative

complexity of the operations. However, these improvements are minor compared to the "tower fields approach" that is going to be analyzed in the following sections.

4.2.3.1 Standard Implementation of AES S-Box

The irreducible polynomial used by the Rijndael S-Box is:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{4.14}$$

An efficient way to calculate the $254-th$ power of each element, is to perform seven squarings and then multiply all the resulting elements.

$$\alpha^{254} = \prod_{i=1}^{7} \alpha^{2^i} = \alpha^{\sum_{i=1}^{7} 2^i} \tag{4.15}$$

The result after every squaring will be a polynomial of degree at most 14, which needs to be reduced in $GF(2^8)$ by using the irreducible polynomial of the S-Box. The exact inversion procedure with the procedure is in the following steps:

- *Representation*
  We represent a general element P of $GF(2^8)$ as a linear polynomial in x with coefficients $p_i, \quad \forall i = 0, ..., 7$ in $GF(2)$, that is $P(x) = p_0 + p_1x + p_2x^2 + p_3x^3 + p_4x^4 + p_5x^5 + p_6x^6 + p_7x^7$. This polynomial representation is going to be used for every byte that is used as input to the S-Box. The polynomial $q(x) = x^8 + x^4 + x^3 + x + 1$ is the AES irreducible polynomial and it is also being used in this representation.

- *Square the polynomial seven times*
  In order to calculate $P^2(x)$, we raise the polynomial $P(x)$ to the second power, that is

  $$\begin{aligned} P^2(x) = p_0^2 + p_1^2x^2 + p_2^2x^4 + p_3^2x^6 &+ p_4^2x^8 + p_5^2x^{10} + p_6^2x^{12} + p_7^2x^{14} = \\ p_0 + c_1x^2 + p_2x^4 + p_3x^6 &+ p_4x^8 + p_5x^{10} + p_6x^{12} + p_7x^{14} \end{aligned}$$

  since $p_i^2 = p_i \in GF(2)\, \forall i = \{0, ..., 7\}$.

  This result is reduced by using the irreducible polynomial q(x), in order to obtain the corresponding element in the field $GF(2^8)$.

  $$\begin{aligned} P^2(x) = p_0 + p_4 + p_6 + p_4x &+ p_6x + p_7x + p_1x^2 + p_5x^2 + \\ p_4x^3 + p_5x^3 + p_6x^3 &+ p_7x^3 + p_2x^4 + p_4x^4 + p_7x^4 \\ +p_5x^5 + p_6x^5 + p_3x^6 &+ p_5x^6 + p_6x^7 + p_7x^7 \end{aligned} \tag{4.16}$$

The final polynomial is also an element of $GF(2^8)$, since it can be written in the following way:

$$W(x) = w_0 + w_1x + w_2x^2 + w_3x^3 + w_4x^4 + w_5x^5 + w_6x^6 + w_7x^7 \quad (4.17)$$

where for the coefficients w_i the following equations hold:

$w_0 = p_0 + p_4 + p_6$	$w_1 = p_4 + p_6 + p_7$	$w_2 = p_1 + p_5$
$w_3 = p_4 + p_5 + p_6 + p_7$	$w_4 = p_2 + p_4 + p_7$	$w_5 = p_5 + p_6$
$w_6 = p_3 + p_5$	$w_7 = p_6 + p_7$	

Table 4.1: The coefficients of $P^2(x)$ after addition

By following a similar procedure, all the powers of $P(x)$ can be obtained, until we reach $P^{128}(x)$. Hereby, the intermediate results are listed:

$$\begin{aligned} P^4(x) = (p_0 + p_2 + p_3 + p_5 + p_6 + p_7) &+ (p_2 + p_3 + p_4 + p_5 + p_6)x + \\ (p_4 + p_5 + p_7)x^2 + (p_2 + p_3 + p_4)x^3 &+ (p_1 + p_2 + p_4 + p_5 + p_6)x^4 + \\ (p_3 + p_6)x^5 + (p_4 + p_7x^6 &+ (p_3 + p_5 + p_6 + p_7)x^7 \end{aligned} \quad (4.18)$$

$$\begin{aligned} P^8(x) = (p_0 + p_1 + p_3) + (p_1 + p_2 + p_3)x &+ (p_2 + p_4 + p_5)x^2 + \\ (p_1 + p_2 + p_6)x^3 + (p_1 + p_2 + p_3 + p_5)x^4 &+ (p_3 + p_4 + p_6 + p_7)x^5 + \\ (p_2 + p_4 + p_6)x^6 + (p_3 + p_4 + p_5 + p_6)x^7 & \end{aligned} \quad (4.19)$$

$$\begin{aligned} P^{16}(x) = (p_0 + p_4 + p_5 + p_6) + p_1x &+ (p_1 + p_2 + p_4 + p_6 + p_7)x^2 + \\ (p_1 + p_3 + p_4 + p_6 + p_7)x^3 + (p_1 + p_5 + p_6)x^4 &+ (p_2 + p_3 + p_7)x^5 + \\ (p_1 + p_2 + p_3 + p_4 + p_7)x^6 &+ (p_2 + p_3 + p_5)x^7 \end{aligned} \quad (4.20)$$

$$\begin{aligned} P^{32}(x) = (p_0 + p_2 + p_3 + p_7) &+ (p_4 + p_6 + p_7)x + (p_1 + p_2 + p_3 + p_7)x^2 + \\ (p_2 + p_3 + p_4 + p_6)x^3 &+ (p_3 + p_4 + p_7)x^4 + (p_1 + p_4)x^5 + x^7 \\ (p_1 + p_2 + p_4 + p_6)x^6 &+ (p_1 + p_4 + p_5 + p_7) \end{aligned} \quad (4.21)$$

$$\begin{aligned} P^{64}(x) = (p_0 + p_1 + p_6) &+ (p_2 + p_3 + p_4 + p_5 + p_6)x + (p_1 + p_6 + p_7)x^2 + \\ (p_1 + p_2 + p_3 + p_5 + p_6)x^3 &+ (p_2 + p_5 + p_7)x^4 + (p_2 + p_6)x^5 + \\ (p_1 + p_2 + p_3 + p_6)x^6 &+ (p_2 + p_5 + p_6 + p_7)x^7 \end{aligned} \quad (4.22)$$

$$\begin{aligned} P^{128}(x) = (p_0 + p_3 + p_5 + p_7) &+ (p_1 + p_2 + p_3)x + (p_3 + p_4 + p_5)x^2 + \\ (p_1 + p_3 + p_6)x^3 &+ (p_1 + p_7)x^4 + (p_1 + p_3)x^5 + \\ (p_1 + p_3 + p_5)x^6 &+ (p_1 + p_3 + p_5 + p_7)x^7 \end{aligned} \quad (4.23)$$

By using the Mathematica functions PolynomialMod[] and PolynomialRemainder[] the previous results have been verified. The Mathematica code for squaring and reducing every polynomial is included in Appendix 1. As we can see from these results, 132 additions are needed. However, some of them appear more than once throughout the whole procedure and can be calculated once and reused several times. After eliminating the common additions, our best construction of a circuit for calculating $p^{128}(x)$ includes **63 XOR gates**. It is assumed that adders with 2-input bits and 1-output bit are used.

- *Six multiplications to obtain* p^{254}
From equation 4.15, we can see that the polynomials we obtained in the $1st$ step should be multiplied, in order to calculate the inverse of the element p in $GF(2^8)$.

$$\begin{aligned} P^{-1}(x) &= P^{254}(x) = \prod_{i=1}^{7} P^{2^i}(x) \\ &= P^2(x) \cdot P^4(x) \cdot P^8(x) \cdot P^{16}(x) \cdot P^{32}(x) \cdot P^{64}(x) \cdot P^{128}(x) \\ &= P^{\sum_{i=1}^{7} 2^i}(x) \end{aligned} \quad (4.24)$$

By counting how many $GF(2)$ operations are needed for one multiplication, then we can calculate the total number of operations performed to compute the inverse $P^{-1}(x)$. Let $P^2(x) = w_0+w_1x+w_2x^2+w_3x^3+w_4x^4+w_5x^5+w_6x^6+w_7x^7$ and $P^4(x) = v_0 + v_1x + v_2x^2 + v_3x^3 + v_4x^4 + v_5x^5 + v_6x^6 + v_7x^7$ be the two polynomials of the second and fourth power of $P(x)$ after computing the sum of every coefficient. The coefficients $w_i, \quad \forall i = \{0, ..., 7\}$ are calculated in the table 4.1 and the coefficients v_i, $\forall i = \{0, ..., 7\}$ can be calculated in the same way for every polynomial $P^i(x)$. Then, for the multiplication we have the following:

$$\begin{aligned} p^2(x) \cdot p^4(x) &= (w_0 + w_1x + w_2x^2 + w_3x^3 + w_4x^4 + w_5x^5 + w_6x^6 + w_7x^7) \\ \cdot(v_0 + v_1x &+ v_2x^2 + v_3x^3 + v_4x^4 + v_5x^5 + v_6x^6 + v_7x^7) = \\ w_0v_0 + w_0v_1x &+ w_0v_2x^2 + w_0v_3x^3 + w_0v_4x^4 + w_0v_5x^5 + w_0v_6x^6 \\ +w_0v_7x^7 &+ w_1v_0x + ... + w_1v_1x^2 + ... + w_1v_7x^8 + w_2v_0x^2 + \\ w_2v_1x^3 &+ ... + w_2v_7x^9 + ... + w_7v_0x^7 + w_7v_1x^8 + ... + w_7v_7x^{14} \\ = \sum_{i=0}^{7}\sum_{j=0}^{7} w_iv_jx^{i+j} \end{aligned} \quad (4.25)$$

Every polynomial has 8 terms that need to be multiplied with another 8-degree polynomial, so in total 64 multiplications must be performed. The coefficients of every term of the polynomial can be written in the following way:

$$\begin{pmatrix} w_0v_0 & w_0v_1 & w_0v_2 & w_0v_3 & w_0v_4 & w_0v_5 & w_0v_6 & w_0v_7 \\ w_1v_0 & w_1v_1 & w_1v_2 & w_1v_3 & w_1v_4 & w_1v_5 & w_1v_6 & w_1v_7 \\ w_2v_0 & w_2v_1 & w_2v_2 & w_2v_3 & w_2v_4 & w_2v_5 & w_2v_6 & w_2v_7 \\ w_3v_0 & w_3v_1 & w_3v_2 & w_3v_3 & w_3v_4 & w_3v_5 & w_3v_6 & w_3v_7 \\ w_4v_0 & w_4v_1 & w_4v_2 & w_4v_3 & w_4v_4 & w_4v_5 & w_4v_6 & w_4v_7 \\ w_5v_0 & w_5v_1 & w_5v_2 & w_5v_3 & w_5v_4 & w_5v_5 & w_5v_6 & w_5v_7 \\ w_6v_0 & w_6v_1 & w_6v_2 & w_6v_3 & w_6v_4 & w_6v_5 & w_6v_6 & w_6v_7 \\ w_7v_0 & w_7v_1 & w_7v_2 & w_7v_3 & w_7v_4 & w_7v_5 & w_7v_6 & w_7v_7 \end{pmatrix}$$

The result will be a polynomial of degree 14 that needs to be reduced to a 8-degree polynomial using the irreducible polynomial $q(x)$. For each reduction we need 7 XOR operations, which represent the subtractions to reduce one degree per time. In total, for each squaring 64-AND and 7-XOR gates are needed.

Complexity
By using the straightforward method, the following operations must be performed:

- **384 AND gates**: For each multiplication of two binary powers 64 $GF(2)$ multiplications are needed, therefore the product of all the seven binary powers requires $6 * 64 = 384$ AND gates.
- **105 XOR gates**: For the calculation of the binary powers 63 additions are needed. For the reduction of each 14th degree polynomial, 7 additions take place, and therefore $6 * 7 = 42$ XOR gates are needed in total.

If we use the free-XOR gate implementation presented in *Chapter 3*, then the additions will not affect the computational cost, and thus the complexity, of this method. However, this approach still demands a vast amount of hardware resources for the 384 AND gates and it is not preferred for implementations of the S-Box.

4.2.3.2 Improvement of the Standard Implementation using the Karatsuba Algorithm

Karatsuba Algorithm is an efficient multiplication algorithm that was discovered by Anatolii Alexeevitch Karatsuba in 1960 and it is based on the divide and conquer paradigm, which was actually first used for this method [A.A95]. It reduces the multiplication of two n-digit numbers to at most $3n^{log_2^3}$ multiplications and it is

therefore faster than the classical algorithm that requires n^2 single-digit products.

Multiplying two polynomials of the same degree in the classical way in $GF(2^2)$, with irreducible polynomial $x^2 + x + 1$, gives the following:

$$(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd = (ac + ad + bc)x + (bd + ac) \quad (4.26)$$

In this way, 4 multiplications and 3 additions are needed. For large numbers, multiplications are the most expensive operations, therefore it is desirable to reduce them. By using Karatsuba's algorithm to calculate the previous result, we need 3 multiplications and 4 additions in the simple case:

$$\begin{array}{rll} Multiplications: & 1. & (a+b)\cdot(c+d) = ac + ad + bc + bd \\ & 2. & b \cdot d \\ & 3. & a \cdot c \\ Additions: & (1)+(2) & \text{to obtain the coefficient of x}, ac + ad + bc \\ & (2)+(3) & \text{gives the constant term } ac + bd \end{array} \quad (4.27)$$

For the general case, let x, y be n-digit numbers in some base B, which can be written as follows:

$$\begin{array}{rcl} x & = & x_0 + x_1 B + x_2 B^2 + ... + x_{n-1} B^{n-1} = X_0 + X_1 B^m \\ y & = & y_0 + y_1 B + y_2 B^2 + ... + y_{n-1} B^{n-1} = Y_0 + Y_1 B^m \\ \text{where} & & \\ X_0 & = & x_0 + Bx_1 + ... + B^{m-1} x_{m-1}, \\ X_1 & = & x_m + Bx_{m+1} + ... + B^{m-1} x_{n-1} \\ Y_0 & = & y_0 + By_1 + ... + B^{m-1} y_{m-1} \\ Y_1 & = & y_m + By_{m+1} + ... + B^{m-1} y_{n-1} \end{array}$$

$for\ 1 \leq m = 2n.$

For the product xy we have the following:

$$\begin{array}{rcl} x \cdot y = (X_1 B^m + X_0)(Y_1 B^m + Y_0) & = & z_2 B^{2m} + z_1 B^m + z_0 \\ \text{where} \quad z_2 = X_1 Y_1, z_1 & = & X_1 Y_0 + X_0 Y_1, z_0 = X_0 Y_0 \end{array} \quad (4.28)$$

In this way, 4 multiplications and 3 additions are needed. However, using the Karatsuba algorithm, we can calculate the product as:

$$z_2 = X_1 Y_1, z_0 = X_1 Y_0 + X_0 Y_1, z_1 = (X_1 + X_0)(Y_1 + Y_0) - z_2 - z_0 \quad (4.29)$$

The product of the binary powers of the polynomials used in the previous section can be also calculated by using Karatsuba algorithm and the required multiplications will be reduced. Equation 4.25 can be written as follows:

$$\begin{aligned} p^2(x) \cdot p^4(x) &= ((a_0 + a_1x) + (a_2x^2 + a_3x^3) + (a_4x^4 + a_5x^5) + \\ (a_6x^6 + a_7x^7)) \cdot ((b_0 + b_1x) &+ (b_2x^2 + b_3x^3) + (b_4x^4 + b_5x^5) + b_6x^6 + b_7x^7)) = \\ &= (a_0 + a_1x)(b_0 + b_1x) + (a_0 + a_1x)(b_2 + b_3x)x^2 + \\ (a_0 + a_1x)(b_4 + b_5x)x^4 &+ (a_0 + a_1x)(b_6 + b_7x)x^6 + (a_2 + a_3x)(b_0 + b_1x)x^2 \\ +... + (a_2 + a_3x)(b_6 + b_7x)x^8 &+ ... + (a_6 + a_7x)(b_6 + b_7x)x^{12} \end{aligned} \quad (4.30)$$

Complexity

This formula consists of 16 multiplications of the form $(a_i + a_{i+1})(b_j + b_{j+1})x^k$, where x^k is a power of x and the product is similar to the Karatsuba form 4.27. Therefore, for each of these partial products we need 3 elementary multiplications and 4 additions, which gives a total of $3 * 16 = 48$ multiplications and $4 * 16 = 64$ additions for this operation.

Finally, for the calculation of the inverse 4.24, we need

- $6 * 48 =$ **288** multiplications in $GF(2)$
- 63 additions for the calculation of the binary powers in step 1, $6 * 64 = 384$ additions for each multiplication and 42 additions for each reduction, which give a total of **489** additions in $GF(2)$

As expected, the number of multiplications is 25% smaller compared to the 384 multiplications in the straightforward method, while the number of additions is increased

4.2.4 Composite Field Implementation of AES S-Box

In this section, we are going to present the "composite field" or "tower field" approach, which was firstly used by Satoh et al in [SMTM01], and gave the most significant reduction in the number of multiplications needed to calculate the inverse in $GF(((2^2)^2)^2)$ in the standard basis. As we are going to see in the following section, Canright reduced further the total number of required operators and chose to calculate the inverse by using the normal basis representation of the fields. Canright's method inspired Boyar and Peralta to construct the most compact AES S-Box implementation up-to-date. Our proposal constitutes a combination of some ideas of the above mentioned methods with the free-XOR method of Kolesnikov and the

GRR method of Pinkas et al. and it uses the standard representation of Satoh et al. (polynomial basis). Therefore, this section plays a crucial role in understanding the context of the efficient secure computation of the AES S-Box.

By using composite field arithmetic it is possible to achieve a significant reduction in the number of gates that are used to design a circuit for a compact AES S-Box. In [SMTM01] an optimization of the S-Box implementation is achieved by using the composite field $GF(((2^2)^2)^2)$, which is constructed by applying multiple extensions of degree 2 to the field $GF(2)$. This is done, in order to reduce the operations needed to calculate the multiplicative inverses of the elements of the S-Box. This method consists of the following steps:

- *Step 1:* Map all the elements of the field $GF(2^8)$ to the composite field $GF(((2^2)^2)^2)$ by using an isomorphism function δ.
- *Step 2:* Compute the multiplicative inverses over the composite field $GF(((2^2)^2)^2)$.
- *Step 3:* Re-map the results to the initial field $GF(2^8)$ by using the inverse function δ^{-1}.

The composite field $GF(2^8)$ of the previous method is constructed by applying a degree-8 extension to $GF(2)$. From *Chapter 2*, we know that the field $GF(2^8)$ is isomorphic to the field $GF(((2^2)^2)^2)$, since they have the same number of elements 2^8. So we can derive the same field, if we apply multiple degree-2 extensions under a polynomial basis using the appropriate irreducible polynomials, namely:

- For the Galois Field $GF(2^2)$ we use the irreducible polynomial $x^2 + x + 1$.
- For the Galois Field $GF((2^2)^2)$ we use the irreducible polynomial $y^2 + y + \phi$, where $\phi = \{10\}_2$.
- For the Galois field $GF(((2^2)^2)^2)$ we use the irreducible polynomial $z^2 + z + \lambda$, where $\lambda = \{1100\}_2$.

In [IT88] it is shown that for any composite field of the form $GF((2^m)^n$, which is constructed using a degree-n extension after a degree-m extension, the multiplicative inverse of an element can be computed as a combination of operations in the lower field $GF(2^n)$ using the equation:

$$P^{-1} = (P^r)^{-1}P^{r-1}, where \quad r = (2^{nm} - 1)/(2^m - 1) \tag{4.31}$$

For our field $GF(2^8) \cong GF((2^4)^2)$ it is $m = 4, n = 2$, so the equation 4.31 becomes:

$$P^{-1} = (P^{17})^{-1} \cdot P^{16} \tag{4.32}$$

In order to calculate the multiplicative inverse of an $8-bit$ element in the composite field $GF(((2^2)^2)^2)$, we need at first to obtain P^{16} with multiple squarings (the hardware costs for computing 2-powers over Galois fields are very small) and the multiply P by P^{16} over $GF(((2^2)^2)^2)$. The inversion of the element P^{17} is performed in the lower field $GF((2^2)^2)$, since P^{17} is always an element of $GF((2^2)^2)$. This is justified as follows: From Fermat's Little Theorem 2.1.2, we have that: For every finite field $GF(q)$, it holds that $a^q = a \, \forall a \in GF(q)$. So $GF((2^2)^2) \cong GF(16) = \{x \in GF(((2^2)^2)^2) | x^{15} = 1\} \cup \{0\}$. For every element $P \in GF(((2^2)^2)^2)$ we have that $P^{255} = (P^{17})^{15} = 1$, which is always true in $GF(((2^2)^2)^2)$ and shows that P^{17} belongs in $GF((2^2)^2) \, \forall P \in GF(((2^2)^2)^2)$.

In this way, the inversion of an element over $GF(((2^2)^2)^2)$ is reduced to inversion of the element over $GF((2^2)^2)$, which reduces the computational workload of the whole procedure. The required operations for this inversion consist of three multiplications, two additions, one constant multiplication, one squaring and one inversion over $GF((2^2)^2)$. This is depicted in the following figure taken by [SMTM01] :

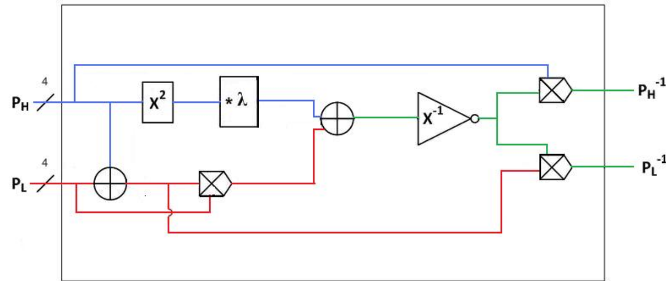

Figure 4.2: Inverter over the composite field $GF(((2^2)^2)^2)$

For the implementation of the S-box, we need to analyze the construction of the inverter over $GF(2^2)^2)^2)$ and we are going to do this recursively. Let $P = P_H z + P_L$ be an element of $GF((2^2)^2)^2)$, with $P_H, P_L \in GF((2^2)^2)$ the 4 most significant and least significant bits respectively. For the representation of the different field elements, we are going to use uppercase Roman letters for elements of $GF(((2^2)^2)^2)$, uppercase Roman letters with indices H, L are going to be used to indicate the 4 highest or lowest bits respectively, for instance $P_H, P_L \in GF((2^2)^2)$. Lowercase Roman letters are going to be used for elements of $GF(2^2)$ and lowercase Greek letters are going to be used for elements of $GF(2)$.

The inverter over $GF(2^2)^2)^2)$ in *Figure 4.2* consists of the following components:

- **Squaring:** The squaring operator is applied on the 4 most significant bits

$P_H = py + q$ with $\alpha, \beta \in GF(2^2)$. The result of this operation is:

$$\begin{aligned} P_H^2 = (py+q)^2 = p^2y^2 + q^2 &= (\alpha x + \beta)^2(y+\phi) + (\gamma x + \delta)^2 = \\ (\alpha x^2 + \beta)(y+x) + (\gamma x^2 + \delta) &= (\alpha(x+1) + \beta)(y+x) + (\gamma(x+1) + \delta) \\ = \alpha xy + \alpha y + \beta y + \alpha x^2 &+ (\alpha + \beta + \gamma)x + \gamma + \delta \\ = (\alpha x + \alpha \oplus \beta)y + ((\beta \oplus \gamma)x &+ \alpha \oplus \gamma \oplus \delta) \end{aligned} \tag{4.33}$$

These calculations can be depicted in the following figure:

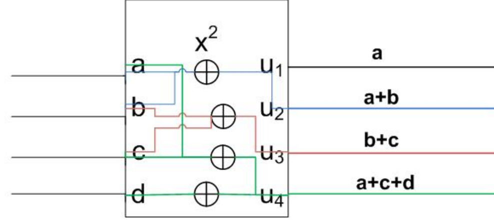

Figure 4.3: Squaring over the composite field $GF((2^2)^2)$

From the above calculations and the figure, **4 XOR operators over** $GF(2)$ are needed to compute the squaring of an element in $GF((2^2)^2)$.

- **Constant multiplier:** After squaring, the 4 most significant bits are multiplied with the constant $\lambda = (x+1)y$, which results in the following:

$$\begin{aligned} P_H \cdot \lambda = (py+q)(x+1)y &= pxy^2 + py^2 + qxy + qy = \\ (\alpha x + \beta)(xy + x + 1) &+ (\alpha x + \beta)(y + x) + (\gamma x + \delta)xy + (\gamma x + \delta)y = \\ ((\beta \oplus \delta)x &+ (\alpha \oplus \beta \oplus \gamma \oplus \delta))y + \alpha x + \beta \end{aligned} \tag{4.34}$$

2 $GF(2)$ **XOR** gates are required. In the previous result there seem to be 4 XOR gates needed for the constant multiplication. If we combine the squaring and the constant multiplication and eliminate the common subexpressions, then we need less additions. This can be justified as follows:

$$\begin{aligned} P_H^2 \cdot \lambda &= (p^2y + q^2)(x+1)y \\ &= ((\alpha x^2 + \beta)(y+x) + (\gamma x^2 + \delta))(xy + y) = \\ \alpha x^2y^2 + \alpha xy^2 &+ \alpha x^2 y + \alpha xy + \alpha xy + \alpha y + \alpha y^2 x + \alpha y^2 + \alpha x^2 y + \\ \alpha xy + \beta y^2 x + \beta y^2 &+ \beta x^2 y + \beta xy + \gamma x^2 y + \gamma xy + \gamma xy + \gamma y + \delta xy + \delta y \\ &= \alpha xy + \alpha x + \alpha + \alpha y + \alpha x + \alpha xy + \beta xy + \beta x^2 + \beta y + \\ \beta x &+ \beta xy + \beta y + \beta xy + \gamma xy + \gamma y + \gamma y + \delta xy + \delta y \\ ((\beta \oplus \gamma \oplus \delta)x &+ (\alpha \oplus \delta))y + \alpha x + \alpha \oplus \beta \end{aligned} \tag{4.35}$$

By combining these two operations, we need only 4 $GF(2)$ XOR gates, which saves 4 XOR gates from the straightforward approach.

- $GF((2^2)^2))$ **Inverter:** The equation 4.32 shows that at this step the inversion will be implemented in the lower field $GF((2^2)^2)$. This inverter is similar to the one over $GF(((2^2)^2)^2)$, whereby 4 bits are inverted instead of 8, and the 2-bits in the beginning should be multiplied with the constant $\phi = x$. As it is depicted in the following figure, a $GF(2^2)$ inverter is used in this case, which is simply reduced to a squaring of the element over $GF(2^2)$. We hereby list

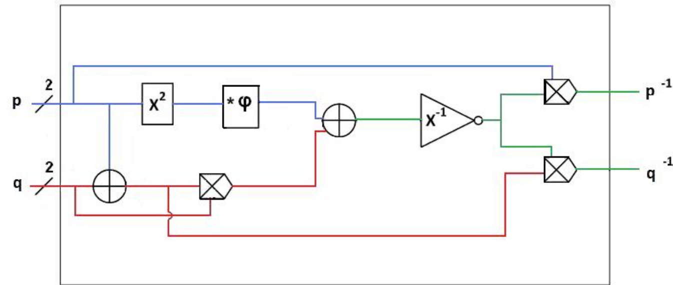

Figure 4.4: Inverter over the composite field $GF((2^2)^2)$

the kind and number of $GF(2)$ operators that take place in the inverter over $GF((2^2)^2)$:

  - *Squaring*: Let $P_H = py + q$ be an element of $GF((2^2)^2)$ representing the most significant bits of the $GF(((2^2)^2)^2)$ element P, where $p = \alpha x + \beta$ and $q = \gamma x + \delta$ are the coefficients in $GF(2^2)$. Squaring over $GF(2^2)$ requires only 1 XOR:

$$p^2 = (\alpha x + \beta)^2 = \alpha x^2 + \beta = \alpha x + (\alpha \oplus \beta) \tag{4.36}$$

  - *Constant multiplication:* The squared element p^2 is multiplied with the constant $\phi = x$:

$$p^2 \cdot x = (\alpha x + \alpha + \beta)x = \alpha x + \alpha + \alpha x + \beta x = (\alpha \oplus \beta)x + \alpha \tag{4.37}$$

    This operation requires 1 XOR between the elements α, β, which is already calculated for the squaring. Therefore, this value can be used also here, and only the coefficients need to be swaped.

  - *Inverter over $GF(2^2)$:* In the relation 2.3, the elements of $GF(2^2)$ are performed and it is noted that the inverse of these elements equal their squaring. Therefore, we have the following:

$$p^{-1} = p^2 = \alpha x + (\alpha \oplus \beta) \tag{4.38}$$

    We need 1 XOR for the inversion over $GF(2^2)$, so inverting is a linear operation in this field. This XOR is going to be calculated before, for the squaring, so we do not need to count it.

- *Addition:* For each addition of two elements over $GF(2^2)$ two $GF(2)$ XOR gates are needed:

$$p \oplus q = (\alpha x + \beta) \oplus (\gamma x + \delta) = (\alpha \oplus \gamma)x + (\beta \oplus \delta) \tag{4.39}$$

  So in total, 4 XOR gates over $GF(2)$ are required for the inversion over $GF(2^2)$.

- *Multiplication:* The Karatsuba algorithm can be directly applied in the multiplication of elements over $GF(2^2)$. Equation 4.27 will give 3 multiplications and 4 additions, which correspond to 3 AND and 4 XOR gates for each $GF(2^2)$ multiplication.

The total number of $GF(2)$ operators that are needed for the inversion over $GF((2^2)^2)$ are **17 additions** and **9 multiplications**.

- $GF((2^2)^2)$ **Addition:** For the addition of two elements over $GF((2^2)^2)$ two $GF(2^2)$ XOR gates are needed, which correspond to four $GF(2)$ XOR gates. Let $P_H = py + q = (ax + \beta)y + (\gamma x + \delta)$, $P_L = ry + s = (\epsilon x + \zeta)y + (\eta x + \theta)$ be the two elements of $GF((2^2)^2)$ that we want to add. Then, we have:

$$\begin{aligned} P_H \oplus P_L = (py + q) \oplus (ry + s) &= (p \oplus r)y + (q \oplus s) = \\ ((ax + \beta) \oplus (\epsilon x + \zeta))y + (\gamma x + \delta) \oplus (\eta x + \theta) &= ((a \oplus \epsilon)x + (\beta \oplus \zeta))y + \\ (\gamma \oplus \eta)x + (\delta \oplus \theta) & \end{aligned} \tag{4.40}$$

  For the inverter two XOR gates are used to add elements over $GF((2^2)^2)$, which give a total of **8 additions** over $GF(2)$.

- $GF((2^2)^2)$ **Multiplication:** This multiplier uses the Karatsuba algorithm to perform the multiplication between 4-bit elements, which is described in the previous section. In this way, the number of required multipliers, that is the number of AND gates, is reduced to 3, and the number of XOR gates is 4. These operations are performed over $GF(2^2)$. For each $GF(2^2)$ multiplier 3 elementary multipliers and 4 additions are needed. In addition, a constant multiplication over $GF(2^2)$ is needed, but this is not an expensive operator as it uses only 1 XOR gate over $GF(2^2)$. These details are depicted in the following figure. The total number of $GF(2^2)$ operators required for one $GF((2^2)^2)$ multiplier is: **9 AND gates** and **21 XOR gates**.

The analysis of this method includes also the generation of the isomorphism functions of Steps 1 and 3. The mappings between the field elements, introduced by the isomorphism functions δ and δ^{-1}, can be described as multiplications of constant matrixes over $GF(2)$. In the description of the inverter, we showed that the constant multipliers in the S-Boxes can be implemented as XOR gates, and therefore, their computation do not influence the complexity of the circuit implementation. For the

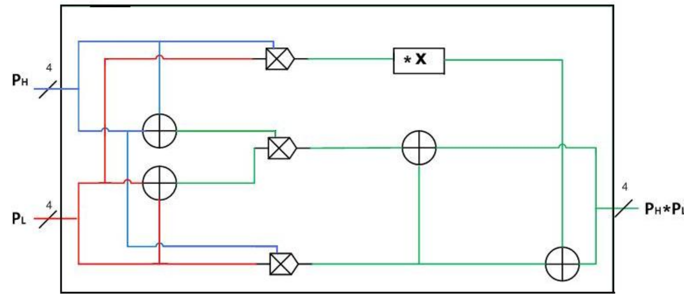

Figure 4.5: Multiplier over the composite field $GF((2^2)^2)$

construction of these functions, we first need to find generator elements $R \in GF(2^8)$ and $S \in GF(((2^2)^2)^2)$, which are both roots of the same irreducible polynomial $q(x) = x^8 + x^4 + x^3 + x^2 + 1$. Then, the definition tables of the isomorphism function can be determined, so that every element in $GF(2^8)$ can be mapped to an equivalent element in $GF(((2^2)^2)^2)$. In [SMTM01], these functions are defined as follows:

$$\delta = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \qquad \delta^{-1} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

At this point it is important to justify, the reason why the isomorphism functions should be used on entering the S-Box and again right after the substitution transformation ends. At first, the isomorphism function δ^{-1} can be merged with the affine transformation $Ax + b$ of the S-Box, in order to further reduce the cost of computing these outputs. As it can be seen in the Mathematica code of *Appendix 2*, after the elements are inverted, they are multiplied with A and then with δ^{-1}, and then the addition of the constant matrix b follows. Since both matrixes A and δ^{-1} are fixed, this matrix multiplication can take place in a preprocessing phase and the resulting matrix can be used directly after the elements are inverted. Moreover, the *MixColumns* step involves multiplications with the constants 2 and 3, which are simple in the standard basis, but not in the subfield basis.

Complexity

Summarizing the results of the previous analysis, for the calculation of the inverse of an element P over the composite field $GF(((2^2)^2)^2)$, we need:

- $3 * 9 = 27$ AND gates to implement the three multipliers and 9 and gates

for the inverter over the lower field, which give a total of **36 elementary multiplications**.

- $3 * 21 = 63$ additions are required for the three multiplications, 8 additions to implement both XOR gates, 17 additions for the inverter over $GF((2^2)^2)$, 4 for the squaring and for the constant multiplication, which give a total of 92 additions in $GF(2)$.

In *Appendix 2* the verification of the gates counting for the composite inverter in Mathematica is given. This Mathematica code computes the inverse recursively over $GF(((2^2)^2)^2)$, which is used to construct the whole S-Box for AES according to the approach followed in [SMTM01]. Actually, when we first computed the number of XOR and AND gates following this approach, the result was 97 XOR gates, but after the observation that the constant multiplication can be merged with the squaring, the number of XOR gates was reduced to 92. At this point, we also note that the number of XOR gates can be further reduced, if we take into consideration that the last two multipliers of the inverter have the same input x^{-1}. Therefore, the XOR gate that adds the two components of the first 4−bits inside the $GF((2^2)^2)$ multiplier, can be used only once and the same output will be used twice. In this way, two XOR gates are saved from the $GF(((2^2)^2)^2)$ inverter and one from the $GF((2^2)^2)$ inverter. The same holds for the input $P_H \oplus P_L$, which is common for the first multiplier and the last one. This observations result in a further reduction in the number of **total XOR gates to 86**.

In the original paper of Satoh et al, it is actually mentioned, that gate reduction is possible by sharing parts of the three $GF((2^2)^2)$ multipliers, where common inputs are used. However, this improvement was not implemented and the number of XORs is not clearly stated in [SMTM01]. Therefore, throughout this thesis, we are going to consider the 86 XOR gates as the best number that can be achieved with this approach.

4.2.5 Composite Fields Approach with Normal Basis

Some improvements on the compact implementation of Satoh et al. were proposed by Canright in [Can05]. These improvements are due to the elimination of common subexpressions, optimization of logic gates and the selection of a different matrix for the isomorphism functions. Canright chose a normal basis representation instead of the polynomial basis for each subfield of $GF(2^8)$. We are going to give a brief description of Canright's improvements, in order to have an insight in the implementation procedure followed later by Boyar and Peralta, who achieved a significant reduction in the number of AND gates by following his approach. However, we are not going to get into details about this method, because the main issue of

our research, the reduction of AND gates in the S-Box inverter, was not essentially achieved by Canright.

We are going to follow the representation for the field elements of the previous section. In this way, a general element P of $GF(2^8)$ can be represented as a linear polynomial in z over $GF(2^4)$, $P = P_H z + P_L$ with coefficients P_H, P_L in the subfield $GF(2^4)$, where the indexes H or L indicate the highest or lowest bits of an element. So the pair $[P_H, P_L]$ represents P in terms of a polynomial basis $[Z, 1]$, where Z is a root of the irreducible polynomial $r(z) = z^2 + T_Z z + N_Z$. Hereby, the indexes of T, N indicates the variable of the field that we are working on. Alternatively, the same element P can be represented using the normal basis $[Z^{16}, Z]$, since it holds $r(z) = z^2 + T_Z z + N_Z = (z + Z)(z + Z^{16})$. So $T_Z = Z + Z^{16}$ is the trace and $N_Z = (Z)(Z^{16})$ is the norm of Z.

An element P_H of $GF(2^4)$ can be represented as a linear polynomial in y with coefficients in $GF(2^2)$, so it can be written as $P_H = py + q$. An irreducible polynomial of this subfield is $s(y) = y^2 + t_y y + n_y$ with $t_y, n_y \in GF(2^2)$ in the polynomial basis and $s(y) = (y + Y)(y + Y^4)$ in the normal basis. Similarly, the field $GF(2^2)$ can be expressed as linear polynomials in x over $GF(2)$, so an element $p \in GF(2^2)$ can be written as $p = \alpha x + \beta$, where $\alpha, \beta \in GF(2)$. In the polynomial basis the irreducible polynomial can be $m(x) = x^2 + x + 1$ and in the normal basis representation $m(x) = (x + X)(x + X^2)$, since the basis will be $[X^2, X]$.

All the operations in $GF(2^8)$ can be expressed in terms of simpler operations in $GF(2^4)$, which can also be expressed as operations in $GF(2^2)$, similarly to the description in the previous section for the case of the polynomial basis. When we use the normal basis, the computation of the multiplicative inverse changes. In $GF(2^8)$, both Z and Z^{16} satisfy the equation $z^2 + T_Z z + N_Z$, where $T_Z = 1 = Z^{16} + Z$. $N_Z = ZZ^{16}$. To simplify the operations, we assume that the trace is unity, so we have $1 = T_Z^{-1}(Z + Z^{16}) = Z + Z^{16}$. More precisely, for the multiplication of two elements, we have the following:

$$(P_H Z^{16} + P_L Z)(Q_H Z^{16} + Q_L Z) = [P_H Q_H + S_H]Z^{16} + [P_L Q_L + S_H]Z \quad (4.41)$$

where $S_H = (P_H + P_L)(Q_H + Q_L)N_Z$.

The inverse of an element can be calculated as follows:

$$(P_H Z^{16} + P_L Z)^{-1} = [S_H^{-1} P_L]Z^{16} + [S_H^{-1} P_H]Z \quad (4.42)$$

where $S_H = P_H P_L + (P_H^2 + P_L^2)N_Z$.

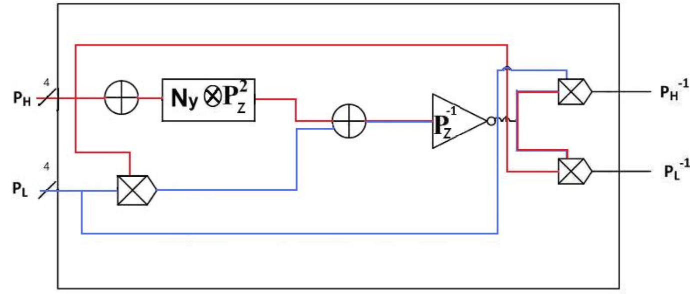

Figure 4.6: Inverter over the field $GF(2^8)$ using the normal basis

The inverter in the normal basis is depicted in *Figure* 4.6.

Inverter in the normal basis: We can see that the inverter using the normal basis is similar to the inverter in *Figure* 4.3, where the polynomial basis is used. Both inverters have the same structure at this level and they use the same number of XOR and AND gates. The only difference is the order of some calculations, since in the normal $GF(2^8)$ inverter the two $GF(2^4)$ elements are firstly XOR-ed bitewise and then raised to the second power and multiplied with the norm, while in the polynomial inverter only the 4 most significant bits are squared and multiplied with the constant term of the irreducible polynomial. Similar calculations should be performed for multiplication and inversion over the subfield $GF(2^4)$ with different arrangement of the operations than in the circuits described for the polynomial basis in the previous section.

One benefit of the normal basis in this stage, is that the inversion over $GF(2^2)$ is the same as squaring, which requires only one swap of the $GF(2)$ elements.

$$(\alpha X^2 + \beta X)^{-1} = (\alpha X^2 + \beta X)^2 = \beta X^2 + \alpha X \tag{4.43}$$

So it is a free operation, while in the polynomial basis squaring needs 1 XOR gate.

Furthermore, the combined operation of squaring and then scaling by the norm N_Z in the normal basis representation is cheaper:

$$\begin{aligned} N_Z \otimes P_H^2 &= N_Z \otimes (pY^4 + qY)^2 = \\ ((p \oplus q)^2)Y^4 + [(N_Y \otimes q)^2]Y &= ((p \oplus q)^2)Y^4 + (N_Y \otimes (\alpha X^2 + \beta X)^2)Y = \\ ((p \oplus q)^2)Y^4 &+ (\alpha X^2 + (\alpha \oplus \beta)X)Y \end{aligned} \tag{4.44}$$

This operation requires 3 XOR gates over $GF(2)$, two for the addition of the elements $p, q \in GF(2^2)$ and one for the addition of the elements $\alpha, \beta \in GF(2)$.

Common Subexpressions: The number of XOR gates can be further reduced by eliminating the common subexpressions. As it is mentioned in *Section* 4.2.4, three XOR gates can be saved from the $GF(((2^2)^2)^2)$ due to the common input x^{-1}. The normal basis inverters over $GF(2^8)$ and $GF(2^4)$ share all three inputs P_H, P_L, P_Z^{-1} among the three multipliers, which leads to saving XOR gates, since the structure of the normal $GF(2^4)$ multiplier is slightly different.

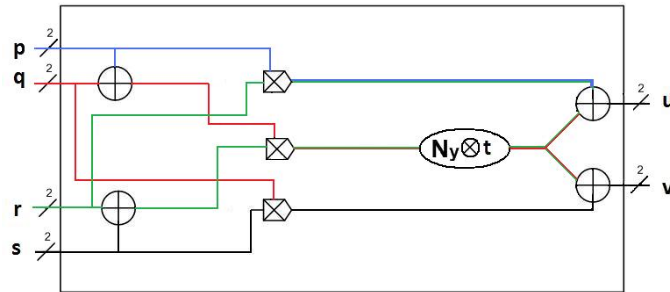

Figure 4.7: Normal multiplier over the field $GF(2^4)$

Therefore, the XOR operation between the subfield elements of each input in the multiplier, can be calculated only once and then reused. For the three common inputs, this gives an advantage of six XOR gates if we use the normal basis representation.

Logic Gate Optimizations: For the $0.13 - \mu m$ CMOS standard cell library, a NAND gate is smaller than an AND gate, so it can be beneficial to consider this logical operation that gives equivalent results. In the $GF(2^2)$ multiplier the AND output bits are combined in pairs and then XORed, that is $[(a \otimes b) \oplus (c \otimes d)]$. This is equivalent to $[(a\, NAND\, b)\; XNOR\, (c\, NAND\, d)]$. The XNOR gate is the same size as the XOR gate, so the latest combination leads to size saving. Furthermore, the XNOR gate can be used for the affine transformation of the S-box, whereby addition with the constant matrix b means applying a NOT to some output bits. This can be done by replacing an XOR gate with an XNOR in the bit-matrix multiply, while leaving the other operations the same. This results in further XOR reductions.

Isomorphism Function: Similar arguments as in *Section* 4.2.4 hold for the use of the isomorphic matrix when entering and right after leaving the S-Box. According to [Can05], the best choice of the matrix for the basis change is the following:

$$X = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

The matrix X is the one that converts the bytes from the subfield basis to the standard basis. So when the input byte gets in the S-Box, we should use the inverted matrix x^{-1} to change into the subfield basis, then compute the multiplicative inverse in the subfield and finally, change back to the standard basis by matrix multiplication with X. Further XOR gates reduction can be done by merging with the affine transformation like before, which means, multiplying every output byte with the product matrix AX, where A is the standard 8×8 matrix of AES for the affine transformation, followed xor-ing with the constant matrix b as usual. In [Can05] it is documented that we can reduce the number of XOR gates further, if we "factor out" the combination of input bits that are shared between different output bits. For the matrix X, this means that for example the result after multiplying rows 2 and 3 with the output bits will include the same sum of the first three bit-multiplications. Therefore, we can calculate this operation once and use it twice, gaining 2 XORs. Another common combination of bits appears between the elements $x_{5,3} + x_{6,3}$ and $x_{5,4} + x_{6,4}$ of the matrix X. In this way, the optimal factoring can be found.

Complexity:

Canright proposed different levels of optimization for the inverter. The following optimizations are comparable to Satoh's approach:

- The hierarchical structure with normal basis needs 88 XOR gates and 36 AND gates, with no improvement for the XOR gates compared to the choice of polynomial basis by Satoh et al.

- The low-level optimizations that came from the elimination of common subexpressions, resulted in 36 **AND gates and** 66 **XOR gates**, which constitutes a significant reduction to the number of additions.

The substitution of 2 XOR gates and a NAND with a NOR gate, gave a further reduction to the number of XOR gates, resulting in 56 XOR, 34 AND and 6 NOR gates. This implementation gives the smallest number of total gates that are needed for the $GF(2^8)$ inverter (with 138 instead of 152 gates that are needed when we

apply the low level optimizations). However, this implementation does not reduce the multiplicative complexity of the circuit, and it is therefore out of the scope of this thesis.

4.2.5.1 A new combinational logic minimization technique

A new combinational logic minimization technique with applications to AES S-Box has been proposed in [BP10]. This technique is based on the observation that circuits with low multiplicative complexity will naturally have large sections which are purely linear, that is contain only XOR gates. Therefore, they followed a two-step process, in order to minimize the computational cost of the AES algorithm:

1. Identify the non-linear components of the circuit and reduce the number of AND gates
2. Find maximal linear components of the circuit and minimize the number of XOR gates that are needed to compute the functions in these linear components.

We are going to analyze the first step of this procedure, since this is the most interesting part for our research. Following the normal basis representation from the previous section, which was introduced by Canright, an element of $GF(2^4)$ can be written as $P_H = (\alpha_1 X + \alpha_2 X^2)Y^2 + (\alpha_3 X + \alpha_4 X^2)Y^8$, where $\alpha_1, \alpha_2, \alpha_3, \alpha_4 \in GF(2)$. The inverse of this element $P_H^{-1} = \beta_1 X + \beta_2 X^2)Y^2 + (\beta_3 X + \beta_4 X^2)Y^8$ can be calculated by using the following polynomials over $GF(2)$:

$$\begin{aligned}
\beta_1 &= \alpha_2\alpha_3\alpha_4 + \alpha_1\alpha_3 + \alpha_2\alpha_3 + \alpha_3 + \alpha_4 \\
\beta_2 &= \alpha_1\alpha_3\alpha_4 + \alpha_1\alpha_3 + \alpha_2\alpha_3 + \alpha_2\alpha_4 + \alpha_4 \\
\beta_3 &= \alpha_1\alpha_2\alpha_4 + \alpha_1\alpha_3 + \alpha_1\alpha_4 + \alpha_1 + \alpha_2 \\
\beta_4 &= \alpha_1\alpha_2\alpha_3 + \alpha_1\alpha_3 + \alpha_1\alpha_4 + \alpha_2\alpha_4 + \alpha_2
\end{aligned} \tag{4.45}$$

The fact that P_H^{-1} is the inverse of P_H can be verified by multiplying the two elements and reducing using the equations: $x^2 = x, x + x = 0, x^2 + x + X^2 = 0, x^2 + x + 1 = 0, Y^4 = Y^2 + X, Y^8 = Y^2 + 1, X^3 = X^2 + X, X^4 = X, X^5 = X^2$.

The main task of this method is to construct a circuit with four inputs and four outputs that calculates the above system of equations using as few multiplications as possible. For this purpose, the following steps were followed:

1. Construct an efficient circuit for one of the equations of the system;
2. Store the intermediate values calculated in the pervious steps and re-use them when the same operations take place;

3. Iterate until all the equations are computed.

Boyar and Peralta used a heuristic, in order to compute the most efficient circuit of each equation in the system, which can be found in [BP09]. They succeeded in determining the multiplicative complexity of all 2^{16} predicates on four bits and it turns out that 3 multiplications are enough to compute any predicate on four variables. They started by computing a circuit for b_4, because they found out that the ordering $\{\beta_4, \beta_2, \beta_1, \beta_3$ gave the best results for this system. The resulting calculations are shown in Table 4.2.

$t_1 = \alpha_1 + \alpha_2$	$t_2 = \alpha_1 \times \alpha_3$	$t_3 = \alpha_4 + t_2$
$t_4 = t_1 \times t_3$	$\beta_4 = \alpha_2 + t_4$	$t_5 = \alpha_3 + \alpha_4$
$t_6 = \alpha_2 + t_2$	$t_7 = t_6 \times t_5$	$\beta_2 = \alpha_4 + t_7$
$t_8 = \alpha_3 + \beta_2$	$t_9 = t_3 \times \beta_2$	$t_{10} = \alpha_4 + t_7$
$\beta_1 = t_{10} + t_8$	$t_{11} = t_3 \times t_{10}$	$t_{12} = \beta_4 \times t_{11}$
$\beta_3 = t_{12} + t_1$		

Table 4.2: Inversion over $GF(2^4)$ using the normal basis

The relevant circuit that represents the above mentioned system of equations is the depicted in *Figure* 4.8.

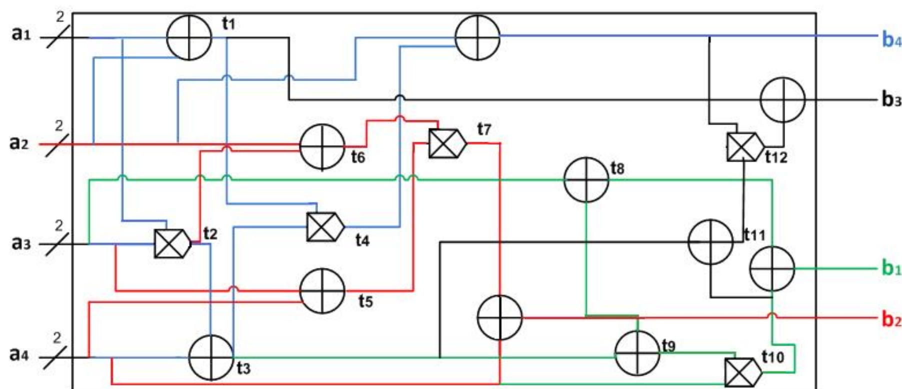

Figure 4.8: The smallest Inverter over the field $GF(2^4)$

This circuit is the most compact $GF(2^4)$ inverter for the AES S-Box that has been constructed so far. It consists of 5 AND gates and 11 XOR gates and it is a significant improvement over previous constructions. Satoh's inverter had 9 AND and 16 XOR gates and Canright's construction consisted of 9 AND and 14 XOR gates, but then he used NAND and NOR gates to optimize the circuit by reaching the number of 8 NAND gates, 2 NOR gates and 9 XOR/XNOR gates.

Complexity:

We have mentioned that Boyar and Peralta used Canright's approach for the inversion over $GF(2^8)$. Therefore, the components of the $GF(2^8)$ inverter are the same as the ones in *Figure* 4.6. Hereby, we are going to find the total number of $GF(2)$ operations of this method by analyzing each component of the circuit.

- For the two additions in the circuit, four XOR gates are needed over $GF(2)$, so in total we have eight XOR gates.
- Squaring an element of $GF(2^4)$ and scaling by the norm N requires 3 XOR gates, as it is justified in the previous section.
- As it is described before, for the inversion over $GF(2^4)$, five AND gates and 11 XOR gates are needed.
- For one multiplication over $GF(2^4)$ 9 AND gates are required, as it is explicitly described in *Section* 4.2.4. We note here, that the three multipliers have three common inputs, P_H, P_L, P_Z^{-1}. So we need 21 XOR gates for the three multiplications.

In total, 32 AND gates and 43 XOR gates are required for the inversion over $GF(2^8)$.

Optimizations for the linear components of the S-Box and merging the affine transformation with the linear parts of the inversion led to further reductions in the total number of XOR gates for the complete S-Box implementation. Boyar and Peralta managed to implement the AES S-Box with a circuit that contains 32 AND gates and 83 XOR/XNOR gates. More details about the linear optimizations and the new heuristic that they introduced can be found in [BP10].

4.2.6 Combinational Logic minimization with Free-XOR Technique

In this section, we are going to present our proposal for a practical implementation of AES in the secure multiparty setting. Our approach constitutes a combination of the tower field approach with the inverse over $GF(2^4)$ of Boyar and the use of the "Free-XOR trick" of Kolesnikov and it involves the smallest circuit that represents the functionality of the AES S-Box in the secure multiparty setting. This idea could be implemented in the following steps:

1. According to the tower fields approach of Paar, it is possible to reduce an 8-bit calculation to several 4-bit ones, and further break-up the 4-bit calculations

into 2-bit ones [Paa94], [FP97]. This approach was efficiently used to construct compact and small binary circuits for the inverter over $GF(2^8) \cong GF(((2^2)^2)^2)$ inside the S-Box of AES. We can use either Satoh's approach with the polynomial basis representation or Canright's approach with the normal basis.

2. For the inverter in the subfield $GF(2^4) \cong GF((2^2)^2)$, we use the combinational logic minimization technique of Boyar and Peralta.

3. Our circuit consists now of 2-to-1 AND and XOR gates. Therefore, for all the XOR gates of the resulting circuit, we can apply the free-XOR technique, more precisely the construction algorithm for garbled circuits, exactly as it is described in *Section 3.1.3*.

4. The rest of the transformations of AES (ShiftRows, MixColumns, AddRoundKey) consist only of bit swaps, constant multiplication of each column with a fixed polynomial and bit-wise XOR operations.

For *Step 2*, it is important to note that the technique to calculate the inverse with only 32 AND gates that was developed in [BP10] for the normal basis approach, can also be applied to the polynomial basis representation. The multiplicative complexity does not depend on the basis we choose for the field representation. A mapping between the two basis can always be found, and in our setting this mapping will be a matrix with binary elements, therefore the transformations will influence only the number of XOR gates.

Hereby, we are going to show that the elements of the $GF((2^2)^2)$ inverter of Satoh et al. can be expressed in a similar way as the elements P_H, P_H^{-1} of Boyar's approach, and they can therefore computed with only 5 multiplications. An element $P_H \in GF((2^2)^2)$ can be represented as a polynomial of y in the form: $P_H = py + q = (\alpha x + \beta)y + (\gamma x + \delta))$, where $p, q \in GF(2^2)$ and $\alpha, \beta, \gamma, \delta \in GF(2)$. Applying the operations of the inverter $GF((2^2)^2)$ in *Figure* 4.4, we have the following:

$$\begin{aligned}
p^2 \cdot x &= (\alpha x + \beta)^2 \cdot x = \beta x + \alpha \\
p \oplus q &= (\alpha \oplus \gamma)x + (\beta \oplus \delta) \\
(p \oplus q) \cdot q &= (\alpha\gamma \oplus \alpha\delta \oplus \beta\gamma \oplus \gamma)x + (\alpha\gamma \oplus \beta\delta \oplus \gamma \oplus \delta) \\
w &= (\alpha\gamma \oplus \alpha\delta \oplus \beta\gamma \oplus \beta \oplus \gamma)x + (\alpha\gamma \oplus \beta\delta \oplus \alpha \oplus \gamma \oplus \delta) \\
x^{-1} = w^{-1} = w^2 &= (\alpha\gamma \oplus \alpha\delta \oplus \beta\gamma \oplus \beta \oplus \gamma)x + (\alpha\delta \oplus \beta\gamma \oplus \beta\delta \oplus \alpha \oplus \beta \oplus \delta) \\
p^{-1} = w^{-1} \cdot p &= (\alpha\beta\gamma \oplus \alpha\beta \oplus \alpha\delta \oplus \alpha \oplus \beta)x + (\alpha\beta\gamma \oplus \alpha\beta\delta \oplus \alpha\delta \oplus \beta\gamma \oplus \beta) \\
q^{-1} = w^{-1} \cdot (p \oplus q) &= (\alpha\beta\gamma \oplus \alpha\gamma\delta \oplus \beta\delta \oplus \alpha \oplus \beta \oplus \gamma)x + (\alpha\beta\gamma \oplus \alpha\gamma\delta \\
&\oplus \alpha\beta\delta \oplus \beta\gamma\delta \oplus \alpha\gamma \oplus \beta\gamma \oplus \alpha\delta \oplus \beta \oplus \gamma \oplus \delta)
\end{aligned} \tag{4.46}$$

From the above mentioned equations, the inverse of the element P_H is:

$$\begin{aligned} P_H^{-1} &= ((\alpha\beta\gamma \oplus \alpha\beta \oplus \alpha\delta \oplus \alpha \oplus \beta)x + (\alpha\beta\gamma \oplus \alpha\beta\delta \oplus \alpha\delta \oplus \beta\gamma \oplus \beta))y \\ &+ (\alpha\beta\gamma \oplus \alpha\gamma\delta \oplus \beta\delta \oplus \alpha \oplus \beta \oplus \gamma)x \\ &+ (\alpha\beta\gamma \oplus \alpha\gamma\delta \oplus \alpha\beta\delta \oplus \beta\gamma\delta \oplus \alpha\gamma \oplus \beta\gamma \oplus \alpha\delta \oplus \beta \oplus \gamma \oplus \delta) \end{aligned} \tag{4.47}$$

Following the notation of *Section* 4.2.5 in equation 4.45, we have the following polynomials that need to be calculated for the inverse:

$$\begin{aligned} b_1 &= \alpha\beta\gamma + \alpha\beta + \alpha\delta + \alpha + \beta = y_1 \\ b_2 &= \alpha\beta\gamma + \alpha\beta\delta + \alpha\delta + \beta\gamma + \beta = b_1 + \alpha\beta\delta + \alpha\beta + \beta\gamma + \alpha = b_1 + y_1 \\ b_3 &= \alpha\beta\gamma + \alpha\gamma\delta + \beta\delta + \alpha + \beta + \gamma = b_1 + \alpha\gamma\delta + \alpha\beta + \alpha\delta + \beta\delta + \gamma \\ &= b_1 + y_3 \\ b_4 &= \alpha\beta\gamma + \alpha\gamma\delta + \alpha\beta\delta + \beta\gamma\delta + \alpha\gamma + \beta\gamma + \alpha\delta + \beta + \gamma + \delta \\ &= b_1 + b_2 + b_3 + \beta\gamma\delta + \alpha\beta + \alpha\gamma + \alpha\delta + \beta\delta + \delta \\ &= b_1 + b_2 + b_3 + y_4 \end{aligned} \tag{4.48}$$

$$\tag{4.49}$$

We can see that all four polynomials have degree 3 and they have the same form as the ones in equation 4.45. After some algebraic operations, we can write the polynomials b_2, b_3, b_4 with only one term of degree 3. The polynomials of this form, which contain only one term of degree three and some remaining terms of degree two (that are used to cancel out equal terms by XORing) are denoted by y_i, $\forall i = \{1, ..., 4\}$. The other terms of degree three, b_1, b_2, b_3, can be calculated in earlier steps and then re-used by XOR-ing the necessary terms. In this way, we can follow a similar factorization procedure to Boyar and Peralta and we can use the same justification of the fact that the multiplicative complexity of inversion is five. One multiplication is needed to compute a polynomial of degree two. Then, an additional multiplication is necessary to produce each of the four polynomials, since each one is of degree three. So in total, five multiplications are needed.

To be more precise, we are going to analyze the factorization procedure that we followed. At first, we have to calculate the following system of equations, which derives from the system 4.48, whereby we have replaced the remaining terms that need to be calculated for every polynomial b_i with y_i $\forall i = \{1, ..., 4\}$:

$$\begin{aligned} y_1 &= \alpha\beta\gamma + \alpha\beta + \alpha\delta + \alpha + \beta \\ y_2 &= \alpha\beta\delta + \alpha\beta + \beta\gamma + \alpha \\ y_3 &= \alpha\gamma\delta + \alpha\beta + \alpha\delta + \beta\delta + \gamma \\ y_4 &= \beta\gamma\delta + \alpha\beta + \alpha\gamma + \alpha\delta + \beta\delta + \delta \end{aligned} \tag{4.50}$$

We computed the most complex term y_4 with two multiplications and we used one multiplication for each of the remaining three terms. The resulted factorization, expressed as a straight-line program over $GF(2)$, is shown in the following table. The outputs are indicated by a star $\star$. By replacing y_1 in equation 4.48, we can

$t_1 = \beta + \gamma$	$t_2 = t_1 + \delta$	$t_3 = \beta \times \delta$	$t_4 = t_3 + \alpha$
$t_5 = t_4 \times t_2$	$t_6 = t_5 + t_3$	$y_4 = t_6 + \delta^\star$	$t_7 = t_2 + \alpha$
$t_8 = t_4 + \beta$	$t_9 = t_7 \times t_8$	$t_{10} = t_9 + t_5$	$t_{11} = t_{10} + t_3$
$y_2 = t_{11} + \beta^\star$	$t_{12} = y_2 + \delta$	$t_{13} = t_{12} + t_3$	$t_{14} = t_{13} \times \alpha$
$y_1 = t_{14} + \beta^\star$	$t_{15} = y_2 + \alpha$	$t_{16} = t_{15} + \gamma$	$t_{17} = y_4 + \delta$
$t_{18} = t_{17} + \alpha$	$t_{19} = t_{18} \times t_{16}$	$t_{20} = t_{19} + y_1$	$t_{21} = t_{20} + \alpha$
$t_{22} = t_{21} + t_3$	$y_3 = t_{22} + t_1^\star$		

Table 4.3: Inversion over $GF(2^4)$ using the standard basis

obtain b_1. Then, by XORing y_2 with b_1, we obtain b_2 and similarly, we can obtain b_3 and b_4 with no additional multiplication. In this way, the inverse over $GF((2^2)^2)$ can be computed.

For *Step 3* of the description of our method, we note that the three assumptions of Kolesnikov's protocol are satisfied. Indeed, the circuits that are constructed according to the previous methods are acyclic boolean circuits with a specified number of gates (211 total number of gates for Satoh, 195 gates for Canright and 115 gates for Boyar and Peralta[1]) and arbitrary fan-out, since the output of each gate was used as input to an arbitrary number of gates. Moreover, all the gates are topologically ordered, since for every gate G_i there are no inputs that are outputs of a successive gate G_j for $j > i$. Finally, we should assume that our method can be applied in the semi-honest case, where the participants follow the protocol, but they try to retrieve more information than they are allowed to.

Complexity: The complexity of our method is equal to the multiplicative complexity of the inversion over $GF(2^8)$, that is $3 * 9 = 27$ AND gates for the three multiplications required for the inversion and 5 AND gates for the computation of the inverse over $GF(2^4)$, which give a total of 32 AND gates. According to the analysis of Satoh et al. in 4.2.4, we need 57 additions for the three multipliers inside the $GF(((2^2)^2)^2)$ inverter and 8 elementary additions for the two XOR gates. For the $GF((2^2)^2)$ inversion in our method we need 21 elementary additions. So in total 96 XOR gates are required for our method.

[1]In order to have comparable results for these three methods, we mention the total number of gates that correspond to the S-Box only, and not to the merged S-Box or the inverse S-Box that were created from Canright and Satoh.

In the secure two-party Yao setting, the XOR gates can be calculated for free and therefore the complexity of the linear components of our circuit does not affect our computations.

4.2.7 Comparison of the proposed Methods

In this final section, we summarize the performance of all the methods that were presented in this chapter. A straightforward comparison between the binary circuit approach of Pinkas et al. and the hardware implementations with composite field arithmetic is not easy to be done, since they are completely different approaches, which have both advantages and disadvantages.

The results obtained by applying the tower field methods show a significant reduction in the number of gates required for the construction of the circuit for inversion in the S-Box of AES compared to the results obtained with the binary circuit approach of Pinkas et al. This reduction is due to the fact that the tower field approach makes use of the arithmetic properties of the inverter of AES over $GF(2^8)$ and reduces the calculations from 8-bit to 4-bit and 2-bit ones.

The following table summarizes the complexity of all the methods that are presented. We included only the AND and XOR gates that are required for the operations in the inverter of the S-Box of AES, in order to have comparable results. The method of Pinkas et al. is not included, since they did not specify the exact number of gates that are required for the S-Box. However, they mention that 33880 gates are required for the implementation of the AES algorithm and there is an estimation of 8639 non-XOR gates that affect the computational workload of the algorithm.

The standard tables of the affine transformation consist of 44 1's, which means that 44 XOR gates are needed to calculate the addition of the corresponding elements. The number of '1' entries are also counted for the isomorphism functions, since the basis transformations should take place when entering and before exiting the S-Box. Usually, this number is reduced by merging the affine and basis transformations. We take into consideration that Satoh et al. and Canright showed that better results can be achieved if we merge the basis transformation matrices with the affine transformations in the S-Box. These improvements are due to elimination of XOR operators only. Boyar and Peralta also gave an optimized way to perform these operations. For instance, Boyar and Peralta performed linear optimizations for the basis transformation matrices and the results in [BP10] show that 53 XOR gates are needed for those operations. However, this number is not explicitly mentioned for every method and it was out of the scope of this thesis to count the minimal number XOR operators. Therefore, we do not include these numbers in the comparison table.

In our approach, we tried to combine the advantages of these approaches, by exploiting the arithmetic properties of AES with a circuit that handles additions and multiplications over the finite field $GF(2^8)$ quicker. By following the tower fields approach, this circuit can be analyzed into binary components, which perform operations over $GF(2)$ in the polynomial basis. Then, by applying the technique of Boyar and Peralta for the $GF((2^2)^2)$ inverter, we managed to obtain the minimal number of AND gates required for this operation. The number of XOR operations in our method is slightly larger than the other approaches, but we focused on reducing the multiplicative complexity of the inversion in the polynomial basis, even if we needed some more XOR gates.

Method	Inverter XOR	Inverter AND	Total Gates	Basis
Standard implementation	105	384	489	Polynomial
Karatsuba	489	288	777	Polynomial
Satoh	86	36	125	Polynomial
Canright	66	36	102	Normal
Boyar	32	32	64	Normal
ours	96	32	128	Polynomial

Table 4.4: Total number of $GF(2)$ operations for the Inverter of the S-Box.

From the comparison table, it is worth mentioning that the focus of the research from Satoh et al., Canright, Boyar and Peralta was mainly on reducing the total number of gates required for the inverter. We focused on reducing the multiplicative complexity of the inverter in the standard basis representation and in this research we managed to obtain better results than Satoh et al. and Canright. We proposed a new architecture, which matched the best AND complexity of Boyar and Peralta for the normal basis, in the polynomial basis representation as well. The number of 32 AND gates for the polynomial basis is what we expected, since the difference is a linear transformation only. Hence, 5 AND gates for the minimal $GF((2^2)^2)$ inverter is not depending on the basis used for the representation of the field elements.

Conclusions

This thesis presented some methods for constructing circuits for the AES algorithm and analyzed extensively the most efficient implementations of the AES S-Box. Satoh et al. took the initiative to construct the inverter of the AES S-Box by using composite field arithmetic. In this way, they created a very compact circuit for AES and they managed to obtain the inverse of an element over $GF(((2^2)^2)^2)$ with only 36 AND gates and 86 XOR gates. Some years later, Canright reduced further the number of required gates for the inverter to 36 AND and 66 XOR gates. This reduction was achieved by applying some optimizations in the choice of logic gates, changing the representation of the field elements from the polynomial to the normal basis and eliminating common subexpressions of the linear components. To the best of our knowledge, the smallest circuit for the inverter in the AES S-Box contains 64 gates (32 AND gates and 32 XOR gates) and was constructed by Boyar and Peralta. Their method requires 115 gates in total (32 AND gates and 83 XOR/XNOR gates) for the whole S-Box. Based on this approach, we showed that this circuit can be constructed in the polynomial basis representation with the same multiplicative complexity and it can be used in the secure two-party setting by applying the "XOR-for-free" technique of Kolesnikov.

In general, we observed that applying the "tower field" approach in boolean circuits can improve their performance and reduce the computational complexity of the operations. By reducing 8-bit calculations to several 4-bit ones and then further to 2-bit ones, we can obtain smaller and more compact circuits that perform the same functionality quicker. Although these optimizations were performed in the setting of minimizing the circuitry of the AES algorithm, they can also be extended in the secure computation of AES. The "XOR-for-free" technique can be applied to reduce further the complexity of the circuit, since the linear components can be evaluated "for free".

A lot of research has been performed in order to determine which is the best choice of the basis representation for these calculations. We note that different basis implementations can give better results only for the linear complexity of the boolean circuits. The multiplicative complexity of such circuits is not affected by different

basis representations, since an isomorphism function between these bases can always be found. This isomorphism function is used to change the basis representations and consists a matrix with binary elements. Therefore, the computations that should be performed correspond to bit-wise additions.

The work of Boyar and Peralta was found at a late stage of the research that was conducted for this thesis. Therefore, we were not able to analyze their work completely and to extend their method to the $GF(2^8)$ inverter for possible further reduction in the number of AND gates. From their work, we conclude that 32 AND gates is probably not optimal for a $GF(2^8)$ inverter, but it will be hard to find such a better circuit, because the heuristic search that they followed will be of high computational complexity and there seems to be no apparent algebraic structure for the optimal circuit for the $GF(2^4)$ inverter.

Bibliography

[A.A95] A.A.Karatsuba. The Complexity of Computations. In *Optimal Control and differential equations*, 1995.

[BCD+08] P. Bogetoft, D. L. Christensen, I. Damgard, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Multiparty Computation Goes Live. Cryptology ePrint Archive, Report 2008/068, 2008. `http://eprint.iacr.org/`.

[BOGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1988. ACM.

[BP09] J. Boyar and R. Peralta. A New Combinational Circuit Optimization and a New Circuit for the S-box for AES. Technical report, Patent application number 61089998 filed with the US Patent and Trademark Office, 2009.

[BP10] J. Boyar and R. Peralta. A New Combinational Logic Minimization Technique with Applications to Cryptology. In *SEA*, pages 178–189, 2010. `http://dx.doi.org/10.1007/978-3-642-13193-6_16/`.

[BR93] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *First ACM Conference on Computer and Communications Se- curity, ACM,*, November 1993.

[Can05] David Canright. A Very Compact S-box for AES. In *CHES*, pages 441–455, 2005. `http://dx.doi.org/10.1007/11545262_32/`.

[CH08] Y. Lindell C. Hazay. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *In Theory of Cryptography Conference - TCC 2008, Springer-Verlag LNCS 4948*, pages 155–175, 2008.

[DK10] I. Damgard and M. Keller. Secure Multiparty AES (short paper). In *Financial Cryptography and Data Security '10, Proceedings of the 14th International Conference*, pages 250–267, 2010.

[EGL85] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commununications of the ACM*, pages 637–647, 1985. volume 28, issue 6.

[FP97] J.L. Fan and C. Paar. On efficient inversion in tower fields of characteristic two. In *Information Theory. 1997. Proceedings., 1997 IEEE International Symposium*, page 20, June 1997.

[Gol04] O. Goldreich. *Foundations of Cryptography, Basic Applications*, volume 2. Cambridge University Press, May 2004.

[GRR98] R. Gennaro, M.O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 101–111, New York, NY, USA, 1998.

[IT88] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *In Information and Computation, volume 78, no. 3*, pages 171–177, 1988.

[Kol05] V. Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. *Advances in Cryptology - ASIACRYPT 2005, LNCS, vol. 3788*, pages 136–155, 2005. Springer, Heidelberg.

[KS08] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR. *35th International Colloquium on Automata, Languages and Programming (ICALP'08), Reykjavik, Iceland, LNCS, vol. 5126*, pages 486–498, 2008. Springer-Verlag, Berlin Heidelberg.

[LN03] R. Lidl and H. Niederreiter. *Finite Fields, Encyclopedia of Mathematcis and its Applications (No.20).* Cambridge University Press, 2003.

[LP04] Y. Lindell and B. Pinkas. A proof of Yao's Protocol for Secure Two-Party Computation. *Journal of Cryptology, Cryptology ePrint Archive, Report 2004/175*, July 2004.

[LP07] Y. Lindell and B. Pinkas. A Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. *Advances in Cryptology - EUROCRYPT 2007, volume 4515*, pages 52–78, 2007.

[MNPS04] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay a secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.

[Paa94] C. Paar. *Efficient VLSI Architectures for Bit-Paraller Computation in Galois Fields*. Phd thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994.

[PSSW09] B. Pinkas, T. Schneider, N.P. Smart, and S.C. Williams. Secure Two-Party Computation Is Practical. In *ASIACRYPT '09: Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan*, pages 250–267, Berlin, Heidelberg, 2009. Springer-Verlag. `http://eprint.iacr.org/`.

[Pub01] Federal Information Processing Standrds Publications. Advanced Encryption Standard. Technical Report FIPS PUB 197, National Institute of Standards and Technology, November 2001.

[Rab81] M. O. Rabin. How to exchange secrets with oblivious transfer. Technical report tr-81, Aiken Computation Lab, Harvard University, May 1981.

[Rij01] V. Rijmen. Efficient Implementation of the Rijndael S-box. 2001. `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/sbox.pdf`.

[Sch05] L.A.M Schoenmakers. *In H.C.A van Tilborg (Ed.), Encyclopedia of Cryptography and Security*, chapter Verifiable Secret Sharing, pages 645–647. New York, Springer, 2005.

[Sha79] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. `http://doi.acm.org/10.1145/359168.359176`.

[SMTM01] A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A Compact Rijndael Hardware Architecture with S-box Optimization. In *ASIACRYPT*, pages 239–254, 2001. `http://link.springer.de/link/service/series/0558/bibs/2248/22480239.htm/`.

[Yao86] A.C. Yao. How to generate and exchange secrets. *In Proceedings of the 27th IEEE Symposium on Foundations of Computer Science (FOCS'86)*, pages 162–167, 1986.

Appendix A

Straightforward Method

Hereby, we present the Mathematica code used to verify the derived equations of the straightforward method for calculating the inverse for the AES algorithm. An element P over $GF(2^8)$ is expressed as a polynomial of degree 7 with coefficients in $GF(2)$. By squaring this polynomial seven times and reducing the result modulo the irreducible AES polynomial $q(x)$, we can obtain the inverse of this element.

($*-$ Straightforward method for calculating the inverse - The coefficients $c0, c1, ..., c7$ belong in $GF(2)$. $*$)

```
p1[x_] = c0 + c1*x + c2*x^2 + c3*x^3 + c4*x^4 + c5*x^5 +
c6*x^6 + c7*x^7;
q[x_] = x^8 + x^4 + x^3 + x + 1;
```

($*p2[x]$ is p1[x] raised to the 2nd power, r2[x] is the remainder after the reduction and we take in mind that $c0, c1, ..., c7$ belong in $GF(2)$, so ${c_i}^2 = c_i \, \forall \, i = 0, ...7$ $*$)

```
p2[x_] = PolynomialMod[Expand[p1[x]^2, Modulus -> 2],
         {q[x],2}];
r2[x_] = Fold[PolynomialRemainder[#1, #2[[1]], #2[[2]],
Modulus ->2] &, p2[x],
{{c0^2 + c0, c0},{c1^2+c1,c1}, {c2^2+c2,c2}, {c3^2+c3,c3},
{c4^2+c4,c4}, {c5^2+c5,c5}, {c6^2+c6,c6}, {c7^2+c7,c7}}]

 c0 + c4 + c6 + c4 x + c6 x + c1 x^2 + c5 x^2 + c4 x^3 +
 c5 x^3 + c6 x^3 + c2 x^4 + c4 x^4 + c5 x^5 + c6 x^5 +
 c3 x^6 + c5 x^6 + c6 x^7 + c7 (x + x^3 + x^4 + x^7)

p4[x_] = PolynomialMod[Expand[p2[x]^2, Modulus -> 2],
         {q[x],2}];
```

```
r4[x_] = Fold[PolynomialRemainder[#1, #2[[1]], #2[[2]],
Modulus ->2] &, p4[x],
{{c0^2+c0, c0}, {c1^2+c1, c1},{c2^2+c2,c2}, {c3^2+c3, c3},
{c4^2+c4, c4}, {c5^2+c5,c5}, {c6^2+c6, c6}, {c7^2+c7, c7}}]

 c0 + c2 + c3 + c5 + c6 + c2 x + c3 x + c4 x + c5 x +
 c6 x + c4 x^2 + c5 x^2 + c2 x^3 + c3 x^3 + c4 x^3 +
 c1 x^4 + c2 x^4 + c4 x^4 + c5 x^4 + c6 x^4 + c3 x^5 +
 c6 x^5 + c4 x^6 + c3 x^7 + c5 x^7 + c6 x^7 +
 c7 (1 + x^2 + x^6 + x^7)

p8[x_] = PolynomialMod[Expand[p4[x]^2, Modulus -> 2],
         {q[x],2}];
r8[x_] = Fold[PolynomialRemainder[#1, #2[[1]], #2[[2]],
Modulus ->2] &, p8[x],
{{c0^2+c0, c0}, {c1^2+c1, c1}, {c2^2+c2,c2}, {c3^2+c3, c3},
{c4^2+c4, c4}, {c5^2+c5,c5}, {c6^2+c6, c6},
{c7^2+c7, c7}}]

 c0 + c1 + c3 + c1 x + c2 x + c3 x + c2 x^2 + c4 x^2 +
 c5 x^2 + c1 x^3 + c2 x^3 + c6 x^3 + c1 x^4 + c2 x^4 +
 c3 x^4 + c5 x^4 + c3 x^5 + c4 x^5 + c6 x^5 + c7 x^5 +
 c2 x^6 + c4 x^6 + c6 x^6 + c3 x^7 + c4 x^7 + c5 x^7 +
 c6 x^7

p16[x_] = PolynomialMod[Expand[p8[x]^2, Modulus -> 2],
          {q[x], 2}];
r16[x_] = Fold[PolynomialRemainder[#1, #2[[1]], #2[[2]],
Modulus->2] &,  p16[x],
{{c0^2+c0, c0}, {c1^2+c1, c1}, {c2^2+c2,c2}, {c3^2+c3, c3},
{c4^2+c4, c4}, {c5^2+c5,c5}, {c6^2+c6, c6}, {c7^2+c7, c7}}]

 c0 + c4 + c5 + c6 + c1 x + c1 x^2 + c2 x^2 + c4 x^2 +
 c6 x^2 + c1 x^3 + c3 x^3 + c4 x^3 + c6 x^3 + c1 x^4 +
 c5 x^4 + c6 x^4 + c2 x^5 + c3 x^5 + c1 x^6 + c2 x^6 +
 c3 x^6 + c4 x^6 + c2 x^7 + c3 x^7 + c5 x^7 +
 c7 (x^2 + x^3 + x^5 + x^6)

p32[x_] = PolynomialMod[Expand[p16[x]^2, Modulus -> 2],
          {q[x], 2}];
r32[x_] = Fold[PolynomialRemainder[#1, #2[[1]], #2[[2]],
Modulus->2] &, p32[x],
```

```
{{c0^2+c0, c0}, {c1^2+c1, c1}, {c2^2+c2, c2}, {c3^2+c3, c3},
{c4^2+c4, c4}, {c5^2+c5, c5}, {c6^2+c6, c6}, {c7^2+c7, c7}}]

 c0 + c2 + c3 + c4 x + c6 x + c1 x^2 + c2 x^2 + c3 x^2 +
 c2 x^3 + c3 x^3 + c4 x^3 + c6 x^3 + c3 x^4 + c4 x^4 +
 c1 x^5 + c4 x^5 + c1 x^6 + c2 x^6 + c4 x^6 + c6 x^6 +
 c1 x^7 + c4 x^7 + c5 x^7 + c7 (1 + x + x^2 + x^4 + x^7)

p64[x_] = PolynomialMod[Expand[p32[x]^2, Modulus -> 2],
          {q[x], 2}]
r64[x\_] = Fold[PolynomialRemainder[#1, #2[[1]], #2[[2]],
Modulus -> 2] &, p64[x],
{{c0^2+c0,c0}, {c1^2+c1,c1}, {c2^2+c2,c2}, {c3^2+c3,c3},
{c4^2+c4,c4}, {c5^2+c5,c5}, {c6^2+c6,c6}, {c7^2+c7,c7}}]

 c0 + c1 + c6 + c2 x + c3 x + c4 x + c5 x + c6 x +
 c1 x^2 + c6 x^2 + c1 x^3 + c2 x^3 + c3 x^3 +
 c5 x^3 + c6 x^3 + c2 x^4 + c5 x^4 + c2 x^5 +
 c6 x^5 + c1 x^6 + c2 x^6 + c3 x^6 + c6 x^6 +
 c2 x^7 + c5 x^7 + c6 x^7 + c7 (x^2 + x^4 + x^7)

p128[x_] = PolynomialMod[Expand[p64[x]^2, Modulus -> 2],
           {q[x], 2}];
r128[x\_] = Fold[PolynomialRemainder[#1, #2[[1]],#2[[2]],
Modulus->2] &, p128[x],
{{c0^2+c0,c0}, {c1^2+c1,c1}, {c2^2+c2,c2}, {c3^2+c3,c3},
{c4^2+c4,c4}, {c5^2+c5,c5}, {c6^2+c6,c6}, {c7^2+c7,c7}}]

 c0 + c3 + c5 + c1 x + c2 x + c3 x + c3 x^2 + c4 x^2 +
 c5 x^2 + c1 x^3 + c3 x^3 + c6 x^3 + c1 x^4 + c1 x^5 +
 c3 x^5 + c1 x^6 + c3 x^6 + c5 x^6 + c1 x^7 + c3 x^7 +
 c5 x^7 + c7 (1 + x^4 + x^7)
```

(* Correctness $p^{256}[x] = p^1[x]$ *)

```
p256[x_] = PolynomialMod[Expand[p128[x]^2, Modulus -> 2],
           {q[x], 2}];
r256[x_] = Fold[PolynomialRemainder[#1, #2[[1]], #2[[2]],
Modulus->2] &, p256[x],
{{c0^2+c0,c0}, {c1^2+ c1,c1}, {c2^2+c2,c2}, {c3^2+c3,c3},
{c4^2+c4,c4}, {c5^2+c5,c5}, {c6^2+c6,c6}, {c7^2+c7,c7}}]

c0 + c1 x + c2 x^2 + c3 x^3 + c4 x^4 + c5 x^5 + c6 x^6 +
```

```
c7 x^7
```

Appendix B

Compact Inverter

The verification of the gates counting for the composite inverter proposed by Satoh et al. without our theoretically justified improvements on the number of XOR gates (from 92 to 86), due to the common inputs to multipliers. The proposed improvement of merging the constant multiplication with the squaring is included in this algorithm.

```
(* Inverter *)

ta = tx = 0; (* number of AND/XOR gates, respectively*)

xor[a_Integer, b_Integer] := Module[, tx++; Mod[a + b, 2]];
xor[a_, b_] := MapThread[xor, a, b];


cmul[a_] := a;
cmul[a_Integer, b_Integer] := xor[a, b], a;
cmulsq[a_] := a; cmulsq[a_Integer, b_Integer] := b, a;
cmulsq[a_,b_,c_,d_]:=xor[b,xor[c,d]],xor[a,d],a,xor[a,b]


mul[a_Integer, b_Integer] := Module[, ta++; Mod[a b, 2]];
mul[a_,b_,c_,d_]:=Module[h,h=mul[b,d];xor[mul[xor[a,b],
                      xor[c,d]],h],xor[h,cmul[mul[a,c]]]];
inv[a_Integer,b_Integer]:=a,xor[a,b];inv[a_,b_]:=
                            Module[s,t, s=xor[a,b];
                            t=inv[xor[cmulsq[a],mul[b,s]]];
                            mul[a,t],mul[t,s]];
```

```
ta
36

tx
92

(* Isomorphism Functions *)

<< FiniteFields`
```

$$\delta = \{\{1,1,0,0,0,0,1,0\},\{0,1,0,0,1,0,1,0\},\{0,1,1,1,1,0,0,1\},\{0,1,1,0,0,0,1,1\},\{0,1,1,1,0,1,0,1\},\{0,0,1,1,0,1,0,1\},\{0,1,1,1,1,0,1,1\},\{0,0,0,0,0,1,0,1\}\};$$
$$inv\delta = \{\{1,0,1,0,1,1,1,0\},\{0,0,0,0,1,1,0,0\},\{0,1,1,1,1,0,0,1\},\{0,1,1,1,1,1,0,0\},\{0,1,1,0,1,1,1,0\},\{0,1,0,0,0,1,1,0\},\{0,0,1,0,0,0,1,0\},\{0,1,0,0,0,1,1,1\}\};$$

```
(* Affine Transformation *)
```

$$A = \{\{1,0,0,0,1,1,1,1\},\{1,1,0,0,0,1,1,1\},\{1,1,1,0,0,0,1,1\},\{1,1,1,1,0,0,0,1\}\{1,1,1,1,1,0,0,0\},\{0,1,1,1,1,1,0,0\},\{0,0,1,1,1,1,1,0\},\{0,0,0,1,1,1,1,1\}\};$$

$$b = \{1,1,0,0,0,1,1,0\};$$

```
(*S-Box*)

For[i = 0, i <= 1, i++,
 For[j = 0, j <= 1, j++,
  For[k = 0, k <= 1, k++,
   iByte = 0, 0, 1, 1, 0, k, j, i;
   iByteprime := iByte[[8]], iByte[[7]], iByte[[6]],
                  iByte[[5]], iByte[[4]], iByte[[3]],
                  iByte[[2]], iByte[[1]];

   c = Mod[δ.iByteprime, 2];
   cp := c[[8]], c[[7]], c[[6]], c[[5]], c[[4]], c[[3]],
         c[[2]], c[[1]];

   a = Flatten[
     inv[cp[[1]], cp[[2]], cp[[3]], cp[[4]],
       cp[[5]], cp[[6]], cp[[7]], cp[[8]]]];
```

```
ap := a[[8]], a[[7]], a[[6]], a[[5]], a[[4]], a[[3]],
       a[[2]], a[[1]];
vf = Mod[A.invδ.ap + b, 2];
sB = vf[[8]], vf[[7]], vf[[6]], vf[[5]], vf[[4]],
      vf[[3]], vf[[2]], vf[[1]];

Print[iByte, BaseForm[FromDigits[sB, 2], 16]]]]]
```

(∗ Output verifying 6 bytes with their corresponding value in hexadecimal format from the S-Box table of the AES specification ∗)

$\{0,0,1,1,0,0,0,0\}$ 4_{16}
$\{0,0,1,1,0,1,0,0\}$ 18_{16}
$\{0,0,1,1,0,0,1,0\}$ 23_{16}
$\{0,0,1,1,0,1,1,0\}$ 5_{16}
$\{0,0,1,1,0,0,0,1\}$ $c7_{16}$
$\{0,0,1,1,0,1,0,1\}$ 96_{16}
$\{0,0,1,1,0,0,1,1\}$ $c3_{16}$
$\{0,0,1,1,0,1,1,1\}$ $9a_{16}$